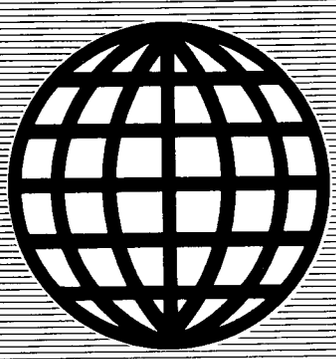


The COMPUTER JOURNAL®

Programming - User Support
Applications

Issue Number 38 May / June 1989 $3.00

C Math
Handling Dollars & Cents With C

Advanced CP/M
Batch Processing and a new ZEX

C Pointers, Arrays and Structures Made Easier
Part 2: Arrays

The Z-System Corner

Information Engineering
The Portable Information Age

Computer Aided Publishing

Shells
ZEX and Hard Disk Backups

Real Computing
The National Semiconductor NS320XX

ZSDOS
Anatomy of an Operating System

ISSN # 0748-9331

THE COMPUTER JOURNAL

Editor/Publisher
Art Carlson

Art Director
Donna Carlson

Circulation
Donna Carlson

Contributing Editors

C. Thomas Hilton
Bill Kibler
Bridger Mitchell
Bruce Morgan
Richard Rodman
Jay Sage
Barry Workman

The Computer Journal is published six times a year by Publishing Consultants, 190 Sullivan Crossroad, Columbia Falls, MT 59912 (406) 257-9119

Entire contents copyright© 1989 by Publishing Consultants.

***Subscription rates**—$16 one year (6 issues), or $28 two years (12 issues) in the U.S., $22 one year in Canada and Mexico, and $24 (surface) for one year in other countries. All funds must be in US dollars on a US bank.*

Send subscriptions, renewals, or address changes to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, Montana, 59912.

Address all editorial and advertising inquiries to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912 phone (406) 257-9119.

The COMPUTER JOURNAL

Issue Number 38 May / June 1989

Editorial 2

C Math

Using Greenleaf's Business MathLib to overcome C's floating point problems.
by Art Carlson 4

Advanced CP/M

Batch processing and a new ZEX.
by Bridger Mitchell 7

C Pointers, Arrays and Structures Made Easier

Part 2: Arrays.
by Clem Pepper 13

The Z-System Corner

Shells and ZEX, high level language Z-System Support, new Z-Node Central, system security under Z-Systems.
by Jay Sage 17

Information Engineering

We will have to revise our software and work habits in order to take advantage of the new portable computers.
by C. Thomas Hilton 21

Computer Aided Publishing

Introduction to publishing and Desk Top Publishing.
by Art Carlson 24

Shells

Zex and hard disk backups.
by Rick Charnes 25

Real Computing

The National Semiconductor NS320XX.
by Richard Rodman 32

ZSDOS—Anatomy of an Operating System

Part 2
by Harold F. Bower and Cameron W. Cotrill 33

Computer Corner *by Bill Kibler* 44

Editor's Page

A Computer Users' Manifesto

The people responsible for the success of the microcomputers were the users who saw the possibilities of what they could accomplish with their own individual system. The existing multiuser multitasking computers were too expensive and too complex—they were run by *computer priests* and only companies with deep pockets could afford them. The users wanted something inexpensive enough so that they could afford it, and simple enough so that they could use it.

The first few microcomputers were only useful to the experimenter and the techie, but when the Apple II appeared, the users became involved. The Apple II+ wasn't much compared with today's standards. It was slow, had anemic drives and limited RAM, but it had BASIC in ROM and it encouraged the user to get involved. Professional business people started writing programs to solve their problems, and the software developers followed with programs designed for the business user.

When the IBM PC appeared, it was also very limited by today's standards, and IBM intended it as a supplement for their corporate accounts, not for the current end user market. But, again, professional business people envisioned what it could do, and developers started writing software for individual user applications.

These two machines are responsible for the micro market of today. The Apple II+ showed what could be done with desktop micros, and the IBM PC gave the micro credibility. It is very important to recognize that the market grew to its current size because of involved professional business users. The growth was not due to large corporations buying micros, and it was not due to large corporations developing huge software products—these came later, they were the effect, not the cause. *Computer Shopper* did not reach its current 600 page size because of large corporate computer users. It was individuals and small businesses which subscribed and bought the products from their advertisers which made it a success.

Today, the marketing emphasis is on complex networked multiuser multitasking systems with enough power to replace the mainframes of a few years ago. It takes a full time professional programmer to work with these systems, in fact they are too much for one person and require a team effort. And, the equipment and implementation efforts are so expensive that only large companies with deep pockets can afford them. It is very similar to what existed before the micros became popular, and we're back to square one again.

We must realize that there are applications for networked multiuser multitasking micros tended by the new generation of computer priests. There are also applications for *plug and play* appliance level systems for non-involved users. These markets are receiving a lot of marketing attention. But, there is also a great demand for products for the involved users, and this demand is not receiving the attention it should.

The Second Computer Revolution

In the first computer revolution, individuals acquired computer power which had been restricted to the establishment. But now, computing power is again being concentrated in large organizations, and it is time to return the power to the people! This will require that we again focus on systems which can be implemented by a knowledgeable end user.

We are not Luddites resisting change, rather we want to guide the change in a different direction. We don't propose going back to machines with 40 column screens and less than 64K of RAM, we want to use all the power available now and in the future. But we want hardware and software with which an individual can implement the system which does what they want.

A professional business person should know more about what is needed for their business than any *computer* consultant they can hire. If they don't, they should hire a *business* consultant to find out what it is that they should be doing. A doctor, dentist, lawyer, baker, hardware store owner, or real estate agent, should be able to design and implement their own application specific programs. Perhaps not the complete accounting program, but certainly the customer, inventory, management, and technical reference applications. Everyone has slightly different needs, and the prepared programs are never quite right. How many dentists are going to be able to program for OS/2 or Windows?

Who is the User?

In a simple case such as a word processor, the user is the one who sits at the keyboard and sees it on the screen. With tools, it gets more complicated.

For example, for a C Windows library which is intended to be incorporated into a C program, is the user the C programmer or is it the programmer's client? In this case there are really two users who must both be satisfied. The programmer is the user of the window programming environment, and it must treat the programmer as a user instead of a techie while the programmer is generating the program. The programmer's client is the user of the completed program, and the library functions must satisfy the client's end user expectations.

Programmers have the fault of designing a program to be used by other programmer techies, even when they intend to sell it to an end user such as a real estate agent. Software should be judged from the viewpoint of its intended audience, not strictly from the viewpoint of another programmer. A programmer should analyze the package for programming defects which could have disastrous results, but an end user should determine if it serves its intended purpose. Software can receive rave reviews from programmers because it uses

all the newest in vogue programming tricks, but it can still be useless for its intended purpose. Great programming does not necessarily mean a great product!

What Does the Future Hold?

We are not yet aware of the significant changes in our lifestyles which will result when personal computers are in widespread use. Before computers are accepted by the general public, people will have to be able to make computers do what they want without feeling challenged, intimidated, or frustrated. The industry has abused the phrase *User Friendly*, but the systems will have to become transparent to the user. The focus has been on learning how to use the tools, but it should be on how to accomplish what the mind conceives!

I feel that consultants should concentrate on assisting small business clients by performing a needs analysis, selecting the hardware and software, and training the client in designing and implementing the application. There will always be clients who want a finished product with no input on their part; but clients should be encouraged to become involved as much as possible, with the consultant there to advise and to take care of any portions which the client can not handle. Of course, the consultant will also service those clients who would rather just write a check and have the completed project dropped in their lap.

Hardware has advanced far beyond the capabilities of the software. Now, we must develop the software which can utilize the full hardware capabilities, and yet which can be fully implemented by professional business people. This is a real challenge with great potential rewards for the successful players.

Data Portability

The computer industry is depending on sales for business computer applications, but the products do not satisfy the business users. There are many areas which must be improved, but data portability is one of the most critical.

Most installations have been made with the intention that all the operations will be performed on the one system. This is a carry-over from mainframes, but now with computers on every desktop, the users expect to share resources with other systems. Our consulting work brings us in contact with a lot of the problems, and the two areas which generate the most complaints are wordprocessor files and data base data files.

A good example is where one of our clients wanted some important letters run off on a friend's daisy wheel printer instead of his dot matrix printer. They were both using IBM clones and 360K disks, but they had different word processors and could not use the same text file. They figured out that the easiest solution was to move the daisy wheel printer, but the originating system would not talk to the printer. That's when they called me.

I asked a few questions, then explained that the dot matrix printer had a parallel interface while the daisy wheel had a serial interface. They became very confused when I tried to explain how they could use the MODE command to change to the serial interface and decided that they needed the file translated. In some cases files can be transferred between wordprocessors through the use of an ASCII file, but in this case it wouldn't work because it included boldfacing. The presence of boldfacing also ruled out writing the print file out to disk and sending it to the printer on the second unit as a binary file transfer because it would include incorrect printer escape codes.

I am currently getting a lot of requests for text file translations, and this gave me another chance to try Timeworks *Beyond Word Writer*. It did the job fast and easy, and I like their user interface – in fact I like the whole software package and am taking a very serious look at it for my primary word processor. It provides a lot of features without overwhelming the user or getting in the way. The entire wordprocessor including the file exchange program sells for $199.95, which is not much more than some programs which only translate files. Contact Timeworks, Inc., 444 Lake Cook Road, Deerfield, IL 60015, phone (312) 948-9200. Their *Publish It!* and *Publish It Light!* desktop publishing programs are also very attractive for someone who does not need the power and complexity of PageMaker.

Another translation program is *Word For Word* by Design Software, but they won't answer my phone calls so I suggest that you avoid that product.

Database files present some very serious problems. Any worthwhile database program must provide some means for importing or exporting data, but I often find systems which have no provisions for data transfer. The developers apparently feel that the user will have to come back to them for any future needs. I can understand that the developer needs to control input in order to protect the consistency of the database. There is no excuse for not providing for output in a standard disk file where the user may want special purpose output from a third party for laser output, cheshire labels processed for third class bulk mail, or to merge names with a outside list for mailing – or even to move the date to a different database system.

End users have much different expectations than computer experts. The users expect transportability between programs and systems. They are starting to flex their muscles and to demand what they need, rather that accepting whatever the experts offer. Don't be surprised to see this tested in court within the next few years.

MAC<-->PC

The widespread use of DTP has resulted in a demand for transferring files between the Mac and the PC. Many of the people involved do not have access to modems, and they need to have their wordprocessor text files moved from one system to the other.

There are a number of approaches to the problem. The most obvious solution is to have both machines and have them communicate. But, most of us can't justify buying the second machine (and learning to use it) solely for file transfer purposes. Another choice is connecting with a modem. But, in this rather isolated area it is difficult to find someone with the desired machine who has a modem. It could also require a several hour round trip to go get the disk after transfer.

I felt that the best solution for my situation was to enable addressing MAC disks with my AT Turbo clone which already had a 3.5 inch drive. I contacted Central Point Software (15220 N.W. Greenbrier Pkwy. #200, Beaverton, OR 97006, (503) 690-8090) about their *Copy II PC Deluxe Option Board*, and they supplied a board for evaluation.

The half-length board fits in the AT slot, and is connected to the drive controller and the drives. After loading the disk file the 3.5 drive can be accessed as a MAC drive with commands such as MCOPY, MDIR, etc. The drive still acts like a regular PC drive when used with the usual commands.

I had a few problems formatting and writing MAC disks when I first got the board, but the drive was flaky and had been giving trouble as a regular AT drive. I replaced the drive with a Teac, and that solved the problems for both AT and MAC disks.

(Continued on page 40)

C Math

Handling Dollars and Cents With C

by Art Carlson

I doubt that many people are as naive as I was when I got my first computer in 1982 and started programming in BASIC. I believed that what was displayed on the screen or sent to the printer was the same as what was stored in the computer. I also believed that when I printed a column of numbers plus their total, that the total would agree with the sum of the column.

I was shocked when I discovered that sometimes things added up, and other times they didn't. I soon narrowed it down to the fact that calculations done with whole numbers (such as 2, 34, 10, etc.) were correct, but that calculations done with decimal fractions (such as 2.14, 5.34, 6.83, etc.) were frequently in error in the last place. That's when I learned about integer (the whole numbers) and floating point (the decimal fractions) math.

I have chosen C used in conjunction with Bytel's Genifer (an application generator) and WordTech's dBXL and Quicksilver for business applications. This combination gives me the ability to do almost anything that I need to do, but I have been concerned about the the floating point errors when using C for financial calculations. The project was at a standstill because I hesitated to spend the effort required to write a C library for financial calculations. So, when Greenleaf Software announced their Business MathLib for C, I contacted them for more information. Their library does much more than what I had planned, and now I can proceed with the project. I'll cover the Business MathLib in more detail, but first let's clarify what's wrong with floating point.

The Problem With Floating Point

First, floating point is satisfactory when it is used for its intended purpose, which is scientific and engineering calculations. In this use, we use a mantissa (which is the numbers), and the exponent (which positions the decimal point). For example Avogadro's number (the number of atoms in a gram-molecular weight of a substance) 6.0247E23 means that the decimal place is 23 places to the right of where it is shown. The number 1234 would be shown as 1.234E3. The results of calculations will be acceptable as long as the worker is aware of the number of significant figures—multiplying 12.34 by 24.56 does not equal 303.0704! There can not be more significant figures in the result than there is in the factor with the least number of significant figures. The above result has to be limited to four figures, which is 303.0 or 303.1, depending on whether you truncate or round.

In scientific work, the numbers 100, 100.0, and 100.00 do not necessarily mean the the same amount because they contain different numbers of significant figures. 100 means that the amount is at least 99.5, but less than 100.5. 100.0 means that it is at least 99.95, but less that 100.05. When talking about money, we assume that the amounts are accurate to the second decimal place, and that's where floating point with its rounding and limited accuracy fails.

I have shelves full of programming books, but I have only been able to find one reference to the floating point problems. What follows is the result of reading between the lines and personal experimentation. Any information or references to books which you can provide will be shared with our readers.

Some C implementations maintain only six digits for floating point numbers. The amount 12345.67 would appear as 1.23456E4, or 1.23457E4 if it is rounded. In either case, you have lost track of the pennies, and will lose track of the dimes for amounts over $99,999.99. Other C implementations provide for additional significant figures, which will alleviate this portion of the problem.

C helps make the programmer's life easier by providing for automatic type conversion, but it also makes it easy to get into trouble. Listing 1 was taken from Borland's Turbo C User Guide, and is the one reference I found to floating point problems. If you enter the values 10 and 3, you get the answer 3.000000 instead of the 3.333333 you expected. This is because a and b are integers and the result of a/b is also an integer which is converted to type float when you assign it to ratio. If you change the type of a and b to float (and change the %d %d in scanf to %f %f), you will get the 3.333333 you expected.

LISTING 1

```
/* Floating point example from Turbo C User's manual pg 162 */

#include <stdio.h>

main()
{
        int a,b;
        float ratio;

        printf(''Enter two numbers:   '');
        scanf(''%d %d'',&a,&b);
        ratio = a / b;
        printf(''The ratio is %f \n'',ratio);
}
```

One of the biggest, and most insidious, problems is that the values printed out do not necessarily agree with what the computer has stored in memory. The program in Listing 2 gets two figures from the user, displays the numbers rounded to two places (as they would be for dollars and cents), and displays the sum rounded to two places. If you enter the numbers 3.456 and 4.566, it displays, "3.46 plus 4.57 equals 8.02"—but the correct answer is 8.03. The error is because a and b are rounded by printf for display, but the numbers as entered are retained in memory and then rounded after being added. Borland's Turbo C was used for all the examples in this article, and there are options to truncate instead of rounding (I left it set for rounding).

This article was inspired by material provided by Don Killen, president of Greenleaf Software, with their Business MathLib.

LISTING 2

```
/* A program to check rounding of floating point numbers for
business math */

#include <stdio.h>
float a,b,c;

main()
{
        printf(''Enter two numbers:    '');
        scanf(''%f %f'',&a,&b);
        c = a + b;
        printf(''%5.2f plus %5.2f equals %5.2f\n'',a,b,c);
}
```

BCD Math

A possible alternative to floating point would be to use Binary Coded Decimal (BCD) math. Internally, BCD looks like a sequence of four bit "nibbles" because you can represent the digits 0 to 9 with just four bits. There would be a structure or header of some sort that indicated things like sign, location of the decimal point, and possibly the exponent (although that is mainly used for scientific and engineering applications).

BCD would be more suitable than floating point for financial calculations since numbers can be represented exactly. But, the only BCD C functions that I am aware of were for the BDS CP/M compiler. At one time Borland provided a BCD version of Turbo Pascal, but as far as I know it is not available for the current release.

If I had to design my own math procedures for financial calculations, it would be based on BCD. But, it would be a major undertaking, and I haven't done it because I can't justify or spare the time which it would require.

A Financial Math Library

Don Killen and Jared Levy from Greenleaf started working on a BCD math library, but ended up with a number representation system which they call DECimal (or just DEC). DEC numbers are stored in a structure which contains 17 or 18 decimal digits plus a couple of bytes indicating the position of the decimal point and the sign. The structure is expandable to include an exponent which is not implemented in the current version.

They wrote functions for the usual math operations such as add, subtract, divide, and multiply, with results that are exact, rounded or truncated only if the user requires it. This in itself will be very valuable for C programmers who don't want to waste time reinventing the wheel. But, they went further, and added nearly 190 functions for things like statistics, internal rate of return, bond prices and yield calculations, annuities, actuarial and general business calculations, depreciation, and date calculations. They called it "Business MathLib."

I've been working with the library for several weeks, and I can't think of what I could have done better. I usually find a lot of shortcomings with most products and say, "They had a good idea, if only they had done it right and completed the job." I can't say that about the Business MathLib, and recommend it. More detailed information on how it works follows, but if you program business applications in C, order their library even if you don't read any further.

MathLib Knows How to Round

Business MathLib gives you a functionality that C by itself doesn't have—the ability to multiply (or add, subtract, etc.) and round at the same time, to any number of decimal places you specify (between 0 and 17). Since you round at the time of the calculation and store the rounded number, subsequent operations on the already rounded numbers will be correct. You can also truncate using a second set of functions if you prefer. The advantage of this is that you can compute things the way they would be in the real world where fractional pennies do not exist.

DEC Number System

The DEC number system is implemented by the structure shown below:

```
typedef struct {
     int attr:
     int id;
     unsigned sl[4];
     short msd;
} IDEC;

typedef struct {
     int lattr;
     int lid;
     unsigned long ls[2];
     short lmsd;
} LDEC;

typedef union {
     IDEC dc;
     LDEC ls;
} DEC;
```

The DEC Structure is actually a union in which the least significant 64 bits of the mantissa can be treated either as four short unsigned integers or two long unsigned integers. In either case, a short signed integer (msd, lmsd) provides the most significant bits. The sign of the DEC number is found in one of the bits in the attr, lattr word. The id, lid word holds a number which indicates the position of the decimal place relative to the

XED4/5/8 Integrated Editor Cross-Assembler

XED4/5/8 is a fast and convenient method to develop and debug small to medium size programs. For use on Z80 machines running Z-system or CP/M. Companion **XDIS4/5/8** disassembler also available.

Targets: 8021, 8022, 8041, 8042, 8044, 8048, 8051, 8052. 8080, Z80, HD64180, and NS455 TMP.

Documentation: 100 page manual.

Features include:

★ Memory resident text (to about 40 KB) for very fast execution. Recognises Z-system's DIR: DU:. Program re-entry with text intact after exit.

★ Built in mnemonic symbols for all 8044,51,52 SFR and bit registers, NS455 TMP video registers and HD64180 I/O ports.

★ Output to disk in straight binary format. Provision to convert into Intel Hex file. Listing to video or printer. A sorted symbol table with value, location, all references to each symbol.

★ Supports most algebraic, logic, unary, and relational operators. Eight levels of conditional assembly. Labels to 31 significant characters.

★ A versatile built in line editor makes editing of individual lines, inserting, deleting text a breeze. Fast search for labels or strings. 20 function keys are user configurable.

★ Text files are loaded, appended, or written to disk in whole or part, any time, any file name. Switchable format to suit most other editors.

★ The assembler may be invoked during editing. Error correction on the fly during assembly, with detailed error and warning messages displayed.

For further information, contact:

PALMTECH
(a division of Palm Mechanical)
cnr. Moonah & Wills Sts.
BOULIA, QLD. 4829
Phone: 6177 463-109 Fax: 6177 463-198 AUSTRALIA

rightmost digit of the mantissa. Normally, results are 64 bits; in many internal calculations 80 bits are calculated to assure a minimum of 64 bits accuracy in the result.

Pointers to DEC Used

DEC numbers are accessed and referred to by pointers. Functions are provided which allocate a DEC or an array of DEC's and return a pointer to the object. All functions in the library take pointers to DEC structures (some take other arguments as well), and most all return a pointer to the resulting DEC number. For example, the addition function setting c = a + b has the form:

```
p = AddDecimal( c, a, b );
```

where a,b,c, and p are all pointers to DEC. If the addition is successful, the destination pointer c is returned. If an overflow or other error should occur, a null pointer is returned, and a global error handler indicates (a) the type of error and (b) the identity of the function that indicated the error.

Conversions

The library provides an extensive set of functions to convert from all C types to DEC and back. Some examples are: ASCII to DEC, ASCII to DEC rounded, DEC to ASCII, DEC to ASCII with commas, DEC to ASCII with commas rounded, DEC to ASCII rounded, DEC to Scientific Notation—there are many more. String formatting includes many options such as the ability to surround negative values with parentheses, fill with leading zeros or dollar signs, etc. For example, to convert a string containing a number to a DEC number, the call is:

```
a = ConvAsciiToDecimal( a, ''1234.56'' );
```

where a is a DEC pointer. The return is null if a conversion error or warning (in case unspecified rounding had to be performed) occurred.

Output

The function for printed output is called PrintDecimal(). It is rather like the familiar printf() except that it knows about DEC (pointer) arguments in addition to all the normal C types. It has many formatting options that are designed to enhance financial and accounting applications. Additionally, it has features which can be linked to other Greenleaf libraries to provide further synergistic effects such as printing negative numbers in different video attributes from positive or zero values. The form of this call is:

```
i = StringPrintDecimal( fmt [,arg] );
```

MathLib Example

The program in Listing 3 is an example of how MathLib can be used to avoid the floating point errors which were encountered with the program in Listing 2. MathLib gives you several choices of when you want to round a number. In this example I used dscanf(), and specified two places with rounding. This way I rounded the numbers on entry, but I could have accepted the numbers as entered and rounded them later with a different function. Now, when you enter the numbers 3.456 and 4.566 you get "3.46 + 4.57 = 8.03" which is the correct answer.

LISTING 3

```
/* Example program using MathLib */

#include <stdio.h>
#include ''gm.h''

main()
{
    DEC da, *a=&da, db, *b=&db, dc, *c=&dc;

    ZeroDecimal(a);
    ZeroDecimal(b);
    ZeroDecimal(c);

    printf(''This program adds two rounded decimal numbers\n'');
    printf(''Enter the first number:   \n'');
    dscanf(''%.2Rt'', a);
    printf(''Enter the second number:   \n'');
    dscanf(''%.2Rt'', b);
    AddDecimal(c, a, b);

    dprintf(''The sum of %t + %t = %t'', a, b, c);
}
```

Summary

Business MathLib would be worth the price if it only offered the four standard math operations with rounding—but it includes much more. Some of the functions (in addition to the rounding, truncation, I/O, and conversion), are: Transcendental (log, trig, power, root, etc.); Test (is DEC positive/negative/equal, etc.); Advanced Business (add/subtract percent, percent change, simple/compound interest, advance payment, loan amortization, discounted cash flow analysis, annuities, bond calculations, depreciation, date manipulation, etc.); Statistics; Linear Estimation.

MathLib works with Greenleaf's DataWindows, SuperFunctions, DataMath, CommLib, and Greenleaf Functions to make a comprehensive business programming package—I'll be looking forward to adding these libraries. The only functions missing are file handling and indexing with file and record locking. Perhaps I'll use the Greenleaf libraries with Raima's dB_VISTA, which provides the missing functions.

Watch for more articles on the Greenleaf products—I'm impressed by their quality and completeness, and don't hesitate to recommend them. Our primary focus will be on defining and solving applications using the tools. Necessary code examples will be included, but the emphasis will be on applications rather than arcane code examples which have little practical use. Borland's Turbo C V2.0 will be used, but the code will be as portable as possible.

The Greenleaf Business MathLib supports Microsoft C v5.0 and 5.1, Lattice C v3.2 and 3.3, and Turbo C v1.5 and 2.0. It is available from most dealers or direct from Greenleaf Software at 16479 Dallas Parkway, Suite 570, Dallas, TX, 75248. Phone 1-800-523-9830 or, from Texas or Alaska, (214)248-2561. The package lists for $325. ■

Advanced CP/M

Batch Processing and a New ZEX

by Bridger Mitchell

More Environmental Programming

Two columns ago I went out on a limb and suggested a number of guidelines for environmentally-sensitive programming—how to write programs that were aware of their host computer's environment, took care to avoid damaging the system, and allowed you to exploit advanced features if they were supported. A number of you have continued the discussion by mail and on the Z-Nodes. There are several areas for further fruitful exploration. I'll touch on one or two this time, and I imagine we can look forward to further exchanges.

Lee Hart asks if there are ways to detect other CPU's (in addition to the HD64180/Z180 and Z280) that support the Z80 instruction set, including the Ferranti, SGS and NEC chips. It would be useful for some programs to know, for example, that they are running on a PC. If you can shed some light on this, please do!

Preserving the Z80 Registers

I (and others) have argued that an environmentally-conscious BIOS will preserve any non-8080 registers that it uses, and restore their values before returning from any BIOS call. Al Hawley and other CP/M veterans recalled that Zilog's early data sheets for the Z80 suggested using the alternate registers to switch contexts in servicing an interrupt.

In an embedded application, using the alternate registers in a service routine is entirely appropriate and efficient, because the designer knows exactly what tasks will use which registers. But it's another matter altogether to use the registers (without preserving them) in an operating system, which is intended to run an *arbitrary* task that may very well itself actively use the same registers.

Unfortunately, several BIOSes were written along the Zilog guidelines, and some authors used the register-swap instructions to save a few bytes. As a result, erratic bugs continue to pop up. Recently, a few users have encountered them when installing ZSDOS, which itself preserves and uses the index registers.

Cam Cotrill has come up with a portable "fix" for part of this defect. It takes the form of a special NZ-COM BIOS segment that saves all of the Z80 registers before every BIOS call and then restores them just before returning. Because NZ-COM allows the user to load a customized BIOS module—in addition to command-processor, named-directory, and other segments—and to adjust its size, it is possible to provide such a band-aid without knowing anything about the hardware features of the BIOS itself! You can find this file in ZSNZBI12.LBR on one of the major Z-Nodes.

As ingenious as this solution is, it would be better still if it were unneeded. And while it handles the BIOS that consumes registers in normal services, it cannot rectify the BIOS service routine that consumes a register for handling interrupts. If you're writing a new BIOS, or have the source to your existing system, take care to preserve the index and alternate registers!

Bridger Mitchell is a co-founder of Plu*Perfect Systems. He's the author of the widely used DateStamper (an automatic, portable file time stamping system for CP/M 2.2); Backgrounder (for Kaypros); BackGrounder ii, a windowing task-switching system for Z80 CP/M 2.2 systems; JetFind, a high-speed string-search utility; DosDisk, an MS-DOS disk emulator that lets CP/M systems use pc disks without file copying; and most recently Z3PLUS, the ZCPR version 3.4 system for CP/M Plus computers.

Bridger can be reached at Plu*Perfect Systems, 410 23rd St., Santa Monica CA 90402, or at (213)-393-6105 (evenings).

Interrupts

How should the environmentally-conscious programmer deal with interrupts? First, a portable program can't use Z80 or 8080/8085 interrupts, because it can't readily determine the availability of interrupt vectors for its own use, and the possible conflicts that could exist with other interrupts used by the system. Therefore, programming interrupt-service routines falls in the province of writing hardware-specific BIOS extensions.

"Programming interrupt-service routines falls in the province of writing hardware-specific BIOS extensions."

The relevant general-purpose guidelines must be limited to procedures for disabling and enabling interrupts. It's rarely necessary to turn off interrupts, and the rule is: keep it short! Interrupts must be disabled whenever your code leaves the system in a state in which an arbitrary interrupt-service routine cannot execute correctly. Keep the stack pointer and system addresses clearly in mind.

Why would you ever use SP for anything but a stack, anyway? Well, it's sometimes a handy way to load a table of word into registers. Repeatedly pushing a constant can be the fastest method of initializing a segment of memory. And several issues ago I described a code-relocation algorithm that used a similar trick to fetch, relocate, and store successive words of code in PRL format.

Interrupts should be disabled (with a DI instruction) just before changing the stack pointer to use it for data operations, and re-enabled (EI) as the instruction that immediately follows restoring the stack pointer to a valid stack. If you don't turn off interrupts, and an interrupt occurs, then when the CPU catches the interrupt it will push the current program counter value onto your "stack", clobbering part of your data area.

In some applications it's necessary to change the BIOS or page 0 vectors. It's remotely possible that an interrupt service routine would use one of these vectors (but only if the BIOS is re-entrant). So, a fastidious guideline would use a DI before any multi-instruction code that changes a vector.

This code:

```
ld (vector_address),de
```

is *atomic*—it changes everything necessary in a single executable instruction, one that cannot itself be interrupted. However, using several instructions to storing the low and high bytes of the address, for example:

```
ld      hl,vector_address
di
ld      (hl),e
inc     hl
ld      (hl),d
ei
```

is not atomic. While that sequence of instructions is executing, the state of the system BIOS vector is not well defined. Without the DI instruction, if an interrupt occurs, its service routine could get an invalid address.

I have used the DI/EI instructions without apparent problems. But when I wrote BackGrounder ii I wanted to ensure wide portability, and I worried about a BIOS that did *not* use interrupts and might behave strangely if they were suddenly enabled. This might seem paranoid, and it's probably the case that a number of other programs would not run on such a system. But I was recalling an early experience of trying to boot an 8085-based S-100 Compupro system in which the interrupt lines had been left floating. When the first EI was executed in the cold- boot code, one of the devices triggered an interrupt, before the BIOS's service routine had been installed.

The routines in Figure 1 can be called in place of inline DI and EI instructions to disable interrupts and conditionally re-enable them. As far as I have been able to determine, this test of the Z80's interrupt status works correctly. However, I have heard reports that some "Z80" CPUs do not report this status correctly. I would welcome any reliable information on this point.

Figure 1. Disable and Re-enable Interrupts

```
;
; Save interrupt status and disable interrupts
;
disable_int:
        push    af              ; save registers
        push    bc
        ld      a,i             ; get interrupt status to A
        push    af
        pop     bc              ; and into C
        ld      a,c             ; and save it
        ld      (intflag),a
        pop     bc
        pop     af
        di                      ; disable non-maskable interrupts
        ret
;
; If interrupts were previously enabled,
; re-enable them.
;
enable_int:
        push    af              ; save register
        ld      a,(intflag)     ; if interrupts
        bit     2,a             ; .. were previously enabled
        jr      z,1$
        ei                      ; ..re-enable them
1$:     pop     af
        ret
```

Batch Processing

Batch processing is running a sequence of commands by submitting a single command to the operating system. In the good old days, the computer operator submitted programs, on 80-column punched cards, to a desk-sized card reader. Programs were batched together by stacking the card decks in a long metal tray. You (or the operator) lugged the tray across the room, crossing your fingers that you didn't trip and spill everything on the floor. Eventually, your job ran and after a seemingly endless wait, the printer disgorged interminable pages of digits, and you went back to debugging yet another core dump. Then the cycle repeated...

CP/M's standard batch processor is the SUBMIT utility. It takes a file of command lines, stored in a file of type SUB, and writes them to a temporary file. The command processor detects this file and gets its commands from it, a line at a time, until it has completed the batch. Then it once again gets its commands from the keyboard.

A submit file, or script, called TEST.SUB might look like this:

```
cmd1 command_tail1
cmd2
cmd3 command_tail3
```

Your command

```
SUBMIT TEST
```

would then cause the three commands to run in sequence.

This basic system works well for programs that require only command-line parameters for their input. But when a program, say CMD1.COM, needs console input, the process stops in its tracks and waits for the user to type in the input. Many times this is just what you want to occur—the user needs to make a real-time decision, and enter data or a response. Often, however, we want the *program input* to also be automated, so that it can be provided from the same script, and the entire batch of jobs will run to completion unattended.

Digital Research, the authors of CP/M, attempted to provide this capability with the XSUB utility. But it was an early attempt to write an RSX (resident system extension), it was buggy, and it proved incompatible with any other RSX.

A major step forward was the development of utilities that combined SUBMIT and XSUB processing, kept the script in memory inside the RSX for faster performance, and supplied a line editor so that short scripts could be typed in on-the-fly when needed. EX.COM was one. Another was the In Memory Submit capability included in Morrow computers which stored the script in banked memory on their CP/M computers.

ZEX

For the Z-System the batch processor has been ZEX—the Z-System EXecutive input processor. It evolved from EX, and has grown like topsy, with significant contributions from Rick Conn, Joe Wright, Jay Sage and others. These increasingly elaborate versions provided for greater control over input, the ability to print messages while the script was running, simple looping, testing of command flow control, etc.

Yet ZEX never quite seemed housebroken, and the tireless Rick Charnes was always coming up with some new batch process that he couldn't quite get ZEX to perform. Moreover, there was no ZEX for Z3PLUS systems. And the hieroglyphics required to write a ZEX script always required relearning just when you needed a quick, automated process.

These warts, and conversations with Joe Wright and Jay Sage, the most recent revisors of ZEX, finally led me to take a fundamental look at this utility. Although the code contained many notable advances, this was truly a "topsy" program, something that had been bandaged and remodeled many times. So, in discussions with Jay, I decided that we need to rethink our objec-

tives and design the program from the outside in. This issue's column focuses on that design, leaving its implementation for another time.

What Should ZEX Be?

The easy part was how it should run. The new ZEX should run on both CP/M 2.2 and CP/M Plus systems. It should be compatible with existing RSX's. It should be able to load and use RSX's as part of a script. And, perhaps, it should be able to invoke a second ZEX script.

These requirements would give us a single batch processor for all Z-Systems, and scripts that could be used on both CP/M 2.2 and CP/M Plus machines without change. A script could be executed while an RSX, such as BackGrounder ii or DosDisk, is already in memory. If needed, the script could load an RSX, for example one to filter printer output.

Preliminary goals for the script language seemed straightforward. The language should allow a standard SUBMIT script to run identically. It should use English-like directives, provide convenient, easily readable comments, and clearly distinguish between input for the command processor, input for programs, and messages and directives. Programs should run identically when the same commands appear in a script, or are typed in at the console.

This is starting to sound like the textbook-prescribed top-down design exercise. As any real programmer knows, that would be a fairy tale, because it seems that all of us just *have to* write some code, if only to check out an idea.

Well, writing code before the design is completed can indeed be productive—the key thing is to avoid getting enmeshed in the thicket of small details before the major skeleton of the project, and possible alternatives, have been sketched and evaluated. So, while drafting and redrafting these preliminary specifications, I also found myself experimenting experimenting with the parsing code, rewriting, modularizing and consolidating several existing ZEX versions, and developing and testing the CP/M Plus interface.

What follows, then, is a still-in-process description of the evolving new ZEX, version 4.0. Your comments and suggestions will be welcome and will surely improve it. I expect ZEX to continue to evolve—it will be easier to add features to the code now that it is more modular. What will require effort is the systematic thought and testing of extensions to the language, to avoid unintended side effects and anomalous cases.

The ZEX Script

ZEX is the Z-System batch-processing language. ZEX.COM is the system tool that implements it. Its purpose is to automate complex and repetitive tasks that require running a series of programs or entering keyboard input.

A ZEX *script* is a text (ASCII) file, or series of text lines entered interactively when ZEX is run. The script file is conventionally given the filetype .ZEX, or sometimes .SUB, for convenience in identifying scripts in a directory.

A script typically consists of a series of *commands* and their *command-tails* that form the input to the ZCPR command processor. In this form the script is equivalent to a CP/M SUBMIT script. In addition, the script may contain *data* for programs that would otherwise be entered from the console keyboard. This feature is similar to, but more advanced than, the CP/M XSUB.

In addition, the ZEX script may contain a number of ZEX *directives* that provide for console messages, waiting for a keypress, ringing the bell, testing command flow control, and so forth.

ZEX explicitly distinguishes between *command-processor* input and *program* input. Normally, ZEX gets all command-line input from the *script* and all program input from the *console*. (This is exactly what SUBMIT does; a SUBMIT script will run identically under ZEX.) But the input sources can be switched by directives. For example, all program input can also be obtained from the script, so that the complete script will run unattended from start to finish.

In reading this, keep clearly in mind the difference between a script file, typed input, and console output. A file is a stream of bytes, broken into lines by a *pair* of bytes: <CR> followed by <LF>. Similarly, when a line of text is output to the screen, it ends with a <CR> (which moves the cursor to the first column of the current line), and a <LF> (which moves it down one line). However, when a line is entered from the keyboard it is terminated by a <CR> only. Thus, in a script you should designate the end of a line of program *input* with a |CR|. For a multi-line message to the screen, terminate each message line with |CRLF|.

The ZEX Language

The ZEX script consists of lines of ASCII text, each terminated by a <CR><LF> pair. (Create the script with a text editor in ASCII (non-document) mode, or just type it into ZEX when prompted.) A number of reserved words, called *directives*, control the various features. Each directive begins and ends with the verticule character '|'. The directives may be entered in upper, lower, or mixed case; we use uppercase here to make them stand out. All script input that is to be sent to a program begins with a '<' character in the first column; all other lines are sent to the command processor or, when specifically directed, are messages sent directly to the console output.

Command-processor input:

- is any line of the script that doesn't begin with '<'.
- is case-independent.
- spaces and tabs at the beginning of a line are ignored.
- is <CR><LF> sensitive. The end of a script line is the end of one command line. Use the |JOIN| directive at the end of a script line to continue the same command line on a second script line. (The <LF> is always discarded).
- use "|NUL| " or |SPACE| to insert a space preceding a command, or after a command and before a comment.
- begin each command (or set of multiple commands, separated by semicolons) on a new script line, optionally preceded or followed by whitespace.
- all whitespace immediately preceding a |JOIN|, and all characters on the line following |JOIN| are discarded.

Program Input:

- is normally obtained from the console.
- begin each line of program input with a '<' in the first column.
- input is case-sensitive.
- data from the script ignores the <CR><LF> at the end of a script line. A single line of program input may spread over several script lines.
- use |CR| to supply a carriage-return.
- use |LF| for linefeed and |CRLF| for carriage-return-linefeed.
- if the program requests more input than is supplied in the script, the remaining input is obtained from the console.
- use |WATCHFOR string| to take further input from the console, until the program sends "string" to the console output, then resume input from the script.

Both:

- use |SAY| to begin text to be printed on the console.
- use |END SAY| to terminate that text.
- use |UNTIL ~| to take further input from the console, until a keyboard ~ is entered. The '~' character may be any character; pick one that won't be needed in entering console input.

• use |UNTIL| to take further input from the console, until a keyboard <CR> is entered.

Comments

A double semicolon ";;" designates the beginning of a comment. The two semicolons, any immediately-preceding whitespace, and all text up to the end of that line of script are ignored.

A left brace '{' in the first column designates the beginning of a comment field; all text, on any number of lines, is ignored up to the first right brace '}'.

Other Directives

Within a directive, a SPACE character is optional. Thus, |IF TRUE| and |IFTRUE| have the identical effect.

```
|IF TRUE|-begin conditional script; do if command flow
    state is true.
|END IF|-end conditional script.
|IF FALSE|-begin conditional script; do if command flow
    state is false.
|RING|-ring console bvll.
|WAIT|-wait until a <CR> is pressed.
|AGAIN|-repeat the entire ZEX script.
|ABORT|-terminate the script if the flow state is true.
|QUIET ON|-turn on the ZCPR quiet flag.
|QUIET OFF|-turn off the ZCPR quiet flag.
|CCPCMD ON|-turn on ZCPR (CCP) command prompt.
|CCPCMD OFF|-turn off ZCPR (CCP) command prompt.
|ZEXCMD ON|-turn on ZEX command prompt.
|ZEXCMD OFF|-turn off ZEX command prompt.
|NUL|-use to make following whitespace significant.
||-same as |NUL|
|SPACE|-one space character.
```

Parameters

ZEX (like SUBMIT) provides for formal parameters designated $0 $1 ... $9. When ZEX is started with a command line such as:

```
A> ZEX SCRIPT1 ARG1 ARG2 ARG3
```

then ZEX reads and compiles the SCRIPT1.ZEX file. In the script, any "$0" will be replaced by "SCRIPT1", any "$1" is replaced by the "first" argument "ARG1", etc.

The script may define "default parameters" for the values $1 ... $9. To do so, enter the three characters "^$n" followed (with no space) by the nth default parameter. When ZEX encounters a formal parameter in the script, it substitutes the command-line parameter, if there was one on the command line, and otherwise the corresponding default parameter, if it was defined.

Alternatively, you can define default parameters by entering "|n = param|", where 'n' is '1' to '9' and "param" is the default string (containing no whitespace).

Control Characters

You enter a control character into the script by entering a caret '^' followed by the control-character letter/symbol. For example, "^A" will enter a Control-A (01 hex). Control- characters may be entered in upper or lower case.

Quotation

ZEX uses a number of characters in special ways: dollar-sign, caret, verticule, left and right curly braces, less-than sign, semicolon, (space, and carriage-return). Sometimes we might want to include these characters as ordinary input, or as output in a screen message. For this, ZEX uses '$' as the *quotation character*. (This is also called the *escape* character, because it allows one to escape from the meaning reserved for a special character.) "Quotation" means that the next character is to be taken literally; I use this term to avoid confusion with the control code 1B hex generated by the *escape key*.

If '$' is followed by any character other than the digits from '0' to '9', that character is taken literally. Thus, if we want a caret in the text and not a control character, we use '$^'. If we want a '<' in the first column of a line that is for the command processor and not for program input, then we use '$<' there instead. And don't forget that if we want a '$' in our script, we must use '$$'. There are some cases, like '$a', where the '$' is not necessary, but it can always be used. To pass a ZEX directive to a program, or the command processor, use the quotation character with the verticule. For example, to echo the string "|RING|", the zex script should be:

```
echo $|RING$|
```

Some Examples

Figure 2 provides several examples of how the new script language should work. You will note a number of differences from the current dialect used, for example, in Rick Charnes' article in this issue. And, no doubt, further improvements will emerge from your suggestions and the actual implementation of the new batch processor.

If You Don't Contribute Anything....

....Then Don't Expect Anything

TCJ is User Supported

Figure 2. ZEX Script Examples

ZEX SCRIPT	INPUT SOURCE/EXECUTION SEQUENCE
cmd1 ;;comment	The CCP receives ''cmd1<cr>''. The spaces before the comment are stripped, and the <cr> at the end of the line is passed to the CCP. The cmd1 program gets its input from the console.
cmd2 |UNTIL|	The CCP receives ''cmd2 '' and then gets additional input from the console, including a <cr>. The cmd2 program gets its input from the console.
|SAY|ccp msg|ENDSAY|cmd3	When the CCP prompts for the next command, ''ccp msg'' is printed on the console. The CCP then receives ''cmd3<cr>''
<text <more text|CR| <new line of text	The cmd3 program gets ''textmore text<cr>new line of text'' If the program requests more input, it comes from the console.
cmd4 cmd4tail <|UNTIL~| <text	The CCP receives ''cmd4 cmd4tail<cr>'' The cmd4 program receives console input until the user types a '~'. Then the program receives ''text'' If the program requests more input, it comes from the console. If the program doesn't use all of the input, it is discarded.
cmd5 |UNTIL~| tail	The CCP receives ''cmd5 '' and then gets additional input from the console, until the user types '~'. The CCP then receives '' tail<cr>''. The program receives input from the console.
|UNTIL|	The CCP receives a command line of input from the console. The program receives input from the console.
|UNTIL| <|SAY|message|ENDSAY| <text	The CCP receives a command line of input from the console. When the program first calls for console input, ''message'' is printed on the console. Then the program receives ''text''. Additional program input is received from the console.
cmd6 <|WATCHFORstring| <|SAY|message|ENDSAY| <text	The CCP gets ''cmd6<cr>'' The cmd6 program gets input from the console, until the characters ''string'' appear at the console output. Then ''message'' appears on the console output, and the program gets ''text''. Further input comes from the console. If ''string'' never appears, all of this is discarded.
alias1 <|UNTIL~| <cmd2text	The CCP gets ''alias1<cr>''. That program, a Z-System alias, puts ''cmd1;cmd2'' into the multiple command line buffer. The CCP then obtains ''cmd1'' from mcl The cmd1 program gets any input from the console. If a '~' is typed, it gets ''cmd2text''. If cmd1 does not request console input, or if no '~' is typed, cmd1 finishes and the CCP then obtains ''cmd2'' from mcl. Assume this case. The cmd2 program obtains input from the console, until a '~' is typed. Then it gets ''cmd2text''. Further input comes from the console
cmd3 <text	The CCP gets ''cmd3<cr>''. The cmd3 program gets ''text''. Further input comes from the console.

■

Hobbyists, Experimenters, R&D Groups

8031 µController Module

Our 8031 µController Module forms the core of a complete prototype and saves hours of hand wiring.

We include a prototype quality pc board populated with an 8031 microprocessor, crystal, 74LS373 Address Latch, capacitors, communications and I/O headers and a 2764 EPROM with diagnostic software. The module also contains a socket for a MAX232 level converter to provide an industry accepted serial communications port if your application requires it.

Documentation is provided and includes programming examples.

$39.95 plus $3.00 S&H per module
Illinois Residents add 6.25% Sales Tax

Cottage Resources
Suite 3-672B, 1405 Stevenson Drive
Springfield, Illinois 62703
(217) 529-7679

Registered Trademarks

It is easy to get in the habit of using company trademarks as generic terms, but these registered trademarks are the property of the respective companies. It is important to acknowledge these trademarks as their property to avoid their losing the rights and the term becoming public property. The following frequently used marks are acknowledged, and we apologize for any we have overlooked.

Apple II, II+, IIc, IIe, Lisa, Macintosch, DOS 3.3, ProDos; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. DateStamper, BackGrounder ii, DosDisk; Plu*Perfect Systems; Clipper, Nantucket; Nantucket, Inc. dBase, dBase II, dBase III, dBase III Plus; Ashton-Tate, Inc. MBASIC, MS-DOS; Microsoft. WordStar; MicroPro International Corp. IBM-PC, XT, and AT, PC-DOS; IBM Corporation. Z80, Z280; Zilog Corporation. Turbo Pascal, Turbo C; Borland International. HD64180; Hitachi America, Ltd. SB180 Micromint, Inc.

Where these, and other, terms are used in The Computer Journal, they are acknowledged to be the property of the respective companies even if not specifically acknowledged in each occurrence.

Plu*Perfect Systems == World-Class Software

BackGrounder ii **$75**

Task-switching ZCPR34. Run 2 programs, cut/paste screen data. Use calculator, notepad, screendump, directory in background. CP/M 2.2 only. Upgrade licensed version for $20.

Z-System **$69.95**

Auto-install Z-System (ZCPR v 3.4). Dynamically change memory use. Order **Z3PLUS** for CP/M Plus, or **NZ-COM** for CP/M 2.2.

JetLDR **$20**

Z-System segment loader for ZRL and absolute files. (included with Z3PLUS and NZ-COM)

ZSDOS ..... **$75, for ZRDOS users just $60**

Built-in file DateStamping. Fast hard-disk warmboots. Menu-guided installation. Enhanced time and date utilities. CP/M 2.2 only.

DosDisk ..... **$30 - $45**

Use MS-DOS disks without copying files. Subdirectories too. Kaypro w/TurboRom, Kaypro w/KayPLUS, MD3, MD11, Xerox 820-I w/Plus 2, ON!, C128 w/1571 -- $30. SB180 w/XBIOS -- $35. Kit -- $45. Kit requires assembly language expertise and BIOS source code.

MULTICPY **$45**

Fast format and copy 90+ 5.25" disk formats. Use disks in foreign formats. Includes DosDisk. Requires Kaypro w/TurboRom.

JetFind **$50**

Fastest possible text search, even in LBR, squeezed, crunched files. Also output to file or printer. Regular expressions.

To order: Specify product, operating system, computer, 5 1/4" disk format. Enclose **check**, adding $3 shipping ($5 foreign) + 6.5% tax in CA. Enclose invoice if upgrading BGii or ZRDOS.

Plu*Perfect Systems
410 23rd St.
Santa Monica, CA 90402

BackGrounder ii ©, **DosDisk** ©, **Z3PLUS** ©, **JetLDR** ©, **JetFind** © Copyright 1986-88 by Bridger Mitchell.

C Pointers, Arrays and Structures Made Easier

Part 2: C Arrays

by Clem Pepper

At the end of the previous segment on pointers we observed the declarations:

```
char *name = ''Jane'';
```

and

```
char name[] = ''Jane'';
```

to be one and the same. You might like to test this using your screen clear function.

Although we don't recognize it as such, a string is actually an array in the sense that its characters occupy sequential locations in memory. To fully appreciate this duality requires an understanding of array principles as employed in C.

Defining the C Array

We use the array when our need is to organize a quantity of our program's data as to make specific data elements readily retrievable. In essence the array is simply a definition for data storage. In the absence of any other direction this storage is in contiguous memory. We might visualize memory as a long line of numbered shoeboxes into which the data is packed. Data, like shoes, requires a variety of sizes. Instead of 11D or 8A data is sized as char, int, float, long and so forth. For most computers a char is a single byte or eight bits. The int is 16 bits, long or float 32. These are fixed for any given system. We can see then why type declarations are essential to the expected execution of our programs.

An array declaration is recognized by the braces following the name: `me_array[n];`. The "n" is the array's size or dimension. That is, the number of elements contained in the array. We may refer to "n" as the array index or subscript. In fact, array numbering is very similar to matrix or determinant subscripting in mathematics. We often see declarations where the brackets are empty. Whether a value is required depends on how the array is initialized.

Hopefully we are familiar with C's storage classes. Variables declared within a function are known only within that function. If the variable is needed in another function a copy is made and passed in the function call. This is known as passing by value. The original value of the variable cannot be modified by the called function. This only applies to variables declared within a function. Global variables are those declared outside a function. Their value is known throughout the program.

The array is an exception to this privacy feature. The array's address is passed when the information it holds is required in a called function. Thus it is possible for the returned value to differ from the original. (Recall the pointer passing in PTR-EX4, Listing 4 which appeared in *TCJ* #37.)

The array variable can be any type: int, char, float, long, double, etc. But only one type for a given array.

Arrays can have more than a single dimension. A two dimensional array could be declared

```
me_2_dim_arr[i][j];
```

And so on. Note the absence of punctuation between the two brackets.

The Integer Array

When we declare a single variable such as:

```
int my_age = 22;
```

the compiler assigns 16 bits somewhere for my__age. Depending on the kind of storage assigned, these 16 bits may be in memory, the stack, or a register.

Now suppose we would like to declare a list of ages for our friends. Joe is 23, Jim 19, Sue 21, Anne 22 for instance. For this we could take advantage of the array and declare:

```
int frnds_age[4];
```

The compiler will now reserve memory for four integer values. The array has a name: `frnds_age`. The name is actually assigned to the first entry in the array, `frnds_age[0]`. It is important to note that the first entry in the array is [0]. Which makes the fourth to be [3].

Now that we have declared our array it is necessary to enter its data. Data is saved in an array in the order in which it is entered. The simplest, most direct method is to enter the data with the declaration. Then the declaration will look like:

```
int frnds_age[4] = { 23,19,21,22 }; /*  { }'s required */
```

We cannot use the declarations

```
int Joe = 23, Jim = 19, Sue = 21, Anne = 22;
```

as array variables even though they are what we have in mind as they do not relate to the array name. Instead:

```
int Joe = frnds_age[0], Jim = frnds_age[1], Sue = frnds_age[2];
int Anne = frnds_age[3];
```

Suppose we write a program based on these values.

```
int my_age = 22;
int frnds_age[4] = { 23,19,21,22 };
int Anne = frnds_age[3];
```

Issue #37 Corrections

The typesetting gremlins struck several times in part one.

First, I lost the closing double quote in the second line of code in the right hand column of page 4. It should read as follows:

```
if(entry != 510) printf(''Sorry 'bout that!'');
```

My typesetting program automatically toggles between opening and closing quotes, and the missing quote caused all the following quotes in the text to be wrong.

I had to reduce the number of spaces for the indenting in order to shorten the line lengths to fit the column width. In some cases I lost track of the indenting level, which makes it difficult to match the start and finish.

```
printf(''My age is %d.\n'',my_age);
printf(''My friends and their ages are:\n'');
printf(''Joe, who is %d; Anne, %d; Sue, %d; and Jim, %d.\n'', \
 frnds_age[0],frnds_age[3],frnds_age[1],frnds_age[2]);
printf(''My favorite friend is Anne who is %d.\n'',Anne);
```

We can list the elements in any desired sequence. Notice how we have declared Anne an integer equal to her age from the array element. The complete program is given in Listing 6 (*Note: In order to avoid confusion when referring to previous listings, the listing numbers continue from issue #37*).

When compiled and run we will see on our screen:

```
My age is 22.
My friends and their ages are:
Joe, who is 23; Anne, 22; Sue, 19; and Jim, 21.
My favorite friend is Anne who is 22.
```

This example illustrates data declared internally within a program and stored in an array for further use. Often there is a need to enter data externally from the keyboard or a disk file for subsequent use within the program. This is illustrated in our next example, ARR_EX2.C (Listing 7).

The program requests the entry of six numbers from the keyboard. Each entry is to be followed by a RETURN. The program will then display the entered value in the array and the square of the entry. Your screen will resemble the following:

```
Enter six numbers between 1 and 127.
Enter digit 1:<cr> 126
Enter digit 2:<cr> 75
Enter digit 3:<cr> 93
Enter digit 4:<cr> 82
Enter digit 5:<cr> 53
Enter digit 6:<cr> 67

i = 0: array[0] = 126
i = 1: array[1] = 75
i = 2: array[2] = 93
i = 3: array[3] = 82
i = 4: array[4] = 53
i = 5: array[5] = 67

i = 0 and *i * *i = 15876
i = 1 and *i * *i = 5625
i = 2 and *i * *i = 8649
i = 3 and *i * *i = 6724
i = 4 and *i * *i = 2809
i = 5 and *i * *i = 4489
```

We see an advantage of pointer variables in their application to finding the squares of the keyboard entries. It is helpful to explore the squaring process as exercised in function sqr__arr__el. The function, taken from the listing, is:

```
/* == Square array content and display on screen == */
sqr_arr_el()
{ int i, sqr_el;
    for(i = 0; i < DIM; ++i)
     {
      parr = &arr_i[i];  /* assign pointer to array  */
      sqr_el = *parr * *parr; /*calculate element square*/
  /* ** Display i and the squared array element ** */
      printf(''i = %d and *i * *i = %d\n'', i,sqr_el);
     }
}
```

As we see, the squaring and display are a looping operation. Each pass of the loop increments a pointer to the next array element. Because we are using pointers there is no need for direct accessing of the element values. In this example we made the global declaration

```
int *parr;
```

Listing 6

```
/*ARR_EX1.C  -------------------- **
** Internally entered data array  */
#include <stdio.h>
#define CLRSCRN ''\033[2J''  /* MS DOS ANSI screen clear */
/* == Begin program == */
main()
{
int my_age = 22;
int frnds_age[4] = { 23,19,21,22 };
int Anne = frnds_age[3];
  puts(CLRSCRN);
  printf(''My age is %d.\n'',my_age);
  printf(''My friends and their ages are:\n'');
  printf(''Joe, who is %d; Anne, %d; Sue, %d; and Jim, %d.\n'', \
   frnds_age[0],frnds_age[3],frnds_age[1],frnds_age[2]);
  printf(''My favorite friend is Anne who is %d.\n'',Anne);
    exit(0);
}
```

Listing 7

```
/* ARR_EX2.C  ---------------------------------- **
** Filling an integer array with keyboard entries  */
#include <stdio.h>
#define CLRSCRN  ''\033[2J''   /* ANSI screen clear */
#define LF ''\n''  /* linefeed */
#define DIM 6    /* array dimension */
int arr_i[DIM];  /* to store externally generated data */
int *parr;       /* pointer to array elements */
/* == Begin program == */
main()
{
    puts(CLRSCRN);                  /* clear the screen  */
     puts(LF);
      fill_arr();       /* fill array and display values  */
       puts(LF);
       sqr_arr_el();  /* square array content and display  */
      exit(0);
}

/* == Fill array from keyboard entries == */
fill_arr()
{ int i;
   printf(''Enter six numbers between 1 and 127.\n'');
    for(i = 0; i < DIM; ++i)
      {
      printf(''Enter digit %d:<cr>'',i+1);
       scanf(''%d'',&arr_i[i]);
      }
     puts(LF);
    for(i = 0; i < DIM; ++i)
      {
  /* ** Display i and keyed value stored in the array ** */
      printf(''i = %d: array[%d] = %d\n'',i,i,arr_i[i]);
      }
}

/* == Square array content and display on screen == */
sqr_arr_el()
{ int i, sqr_el;
    for(i = 0; i < DIM; ++i)
     {
      parr = &arr_i[i];     /* assign pointer to array  */
      sqr_el = *parr * *parr;      /* calculate i exp 2  */
 /* ** Display i and its squared value from the array ** */
      printf(''i = %d and *i * *i = %d\n'', i,sqr_el);
     }
}
```

Listing 8

```
/* ARR_EX3.C  -------------------------- **
** Strings and arrays have a lot in common */

#include <stdio.h>
#define CLRSCRN  ''\033[2J'' /* ANSI screen clear */

  /* === string array using characters. === */
char chr_arr[] = { 'H','i',' ','Y','\'','A','l','l','\0' };
  /* === string array using a character pointer  === */
char *ptr = ''Anything allright?'';

/* == Begin program == */
main()
{ int i;
   puts(CLRSCRN);              /* clear the screen  */

  /* print chr_arr[] ''Hi Y'All'' character by character  */
      for(i=0; i < 9; ++i)  printf(''%c'',chr_arr[i]);

  /* return cursor to left edge of the screen          */
       putchar('\n');

  /* print ''Anything allright?'' using putchar in a    */
  /* while loop. The null (\0) string terminator ends */
  /* the while.                                        */
      while(*ptr) putchar(*ptr++);

  /* return cursor to left edge of the screen          */
       putchar('\n');

  /* replace the present string in ptr with new text  */
      ptr = ''Hey, ya gotta be kidding!'';
  /* ... and display it.                               */
      while(*ptr) putchar(*ptr++);

    exit(0);
}
```

Listing 9

```
/* ARR_EX4.C  ------------------------ **
** Entering disk file text to an array  */
#include <stdio.h>
#define CLRSCRN ''\033[2J''  /* ANSI screen clear */
/* == Begin program == */
main(num,fname)
int num;
char *fname[];
{
 FILE *in;
  char file_in[880];
  int ch, i = 0;
   puts(CLRSCRN);

if(num != 2) { printf(''A source file is required.\n''); exit(0);}
 else if((in = fopen(fname[1],''rb'')) != NULL)
                      {
                     {
   while((ch = getc(in)) != EOF)    /* get char from in */
                     {
     file_in[i++] = ch;
                     }
                       }
          fclose(in);
                      }
 /* ** Convert final char to '\0'; transfer count  ** */
     file_in[i] = '\0';

 /* ** Call subroutine for screen playback ** */
       disp_txt(file_in,i);
        exit(0);
}
```

(Continued)

with no assignment. The assignment takes place when required in the loop operation. A new value is acquired on each pass. The descriptive *i * *i used in printf was merely to obtain a short line length. Interpret it to mean the content of the ith array element times itself.

The Character Array

We have touched on the character array in describing the character pointer. Specifically, we have observed that a string pointer and string array declaration are identical. While it will often be to our advantage to employ the string pointer there are situations in which the array is needed. The most obvious is to address specific characters within the string.

The character array example of Listing 8 may be somewhat confusing at first. That is not the intent, which is to illustrate the versatility available to us when working with character arrays and strings.

Earlier we declared the integer array

```
int frnds_age[4] = { 23,19,21,22 };
```

The character array has a similar declaration. But where we entered integer values in frnds_age we now enter characters instead. A character in C is identified by a single quote (') on each side. The character array for ''Hi Y'All'' appears as the global

```
char chr_arr[] = {'H','i',' ','Y','\'','A','l','l','\0' };
```

We may wonder at the entry '\''. The backslash, \, is required to identify the ' as an element of the array. If the \ is omitted the compiler will declare an error. The \ is required whenever we include another backslash or a double quote also.

Because this array is simply a string in an unfamiliar form the closing null character, '\0,' must be included. When the compiler reads the '\0' it knows the string is ended. Whenever we dimension a character array we must add one count for the closing null.

Another global is

```
char *ptr = ''Anything allright?'';
```

Note that this could just as easily have been declared locally. More importantly, following the printing of its message, a character at a time with putchar, the pointer is re-assigned to a new string and again displayed using a similar loop.

Now let's further expand our expertise by reading in a text file from disk. We can then play it back for display on our screen. If we wish, we can make changes to the text before or during the playback.

Listing 9 describes the program. Our attention will be first drawn to the function main. Where main in previous examples has been followed by empty parentheses, (), we now find the arguments num,fname. Be aware the arguments argc,argv are typically used with main when command line arguments are to be included. However this terminology is not mandatory (with the compilers I have used, at least). I find num and fname more descriptive of what they stand for.

The first argument, num (or argc), reports the number of required entries on the command line. It is declared an integer. The minimum number of entries is 1; our program. We increment num for each filename addition.

The declaration for fname (or argv) is more complex. Note that the declaration reads

```
char *fname[];
```

This declaration is for a pointer to the array fname[]. It can also be written as

```
char **fname;
```

when we think about it.

Our objective with this program is to read a text (ASCII) file from disk and display it on the screen. We will first store the text in an array. We will then read the array element by element for

```
/* == Display read in text on screen == */
disp_txt(text_in,j)
 char text_in[];              /* text array      */
 int j;                       /* final subscript */
{
 int i = 0;
       while(j--)  {
        putc(text_in[i],stdout);
         if(text_in[i+1] == '\r')
       /* add end of line signal */
         putch('~');
          i += 1;   }
}
```

Listing 10

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
So twice five miles of fertile ground
With wall and towers were girdled round.
A savage place! as holy and enchanted
As e'er beneath a waning moon was haunted
By woman wailing for her demon-lover!
```

the screen display. Whenever the program detects a carriage return, '\r', it will insert the tilde (~). The tilde will appear on the screen only, it is not added to the disk file.

In this program we are introduced to a new C type, FILE. The source of FILE is a structure found in the library #include <stdio.h>. A detailed explanation of FILE is beyond the scope of this article. The declaration

```
FILE *in;
```

is a pointer to the disk file to be read in.

Continuing on with the program we find an error detection test. As mentioned, if the program name alone is on the command line, num will have a value of 1. The program requirement is for 2. An error message is printed and the program exits. Listing 10 provides the opening lines from Coleridge's poem, *Xanadu*, as a file you can use with the program. To run the program with this file type the following command line entry

```
ARR_EX4 XANADU
```

and press the Enter (RETURN) key. If XANADU is on a different disk drive than ARR_EX4.EXE, say drive A, simply type

```
ARR_EX4 A:XANADU
```

Of course you can read in any text file you like. But if it exceeds the 880 char dimension you will need to increase the array size and re-compile the program.

The next line

```
else if((in = fopen(fname[1],''rb'')) != NULL)
```

requires some explaining. Let's begin with the function fopen. This call is to open the file named on the command line for reading. The file, fname, is designated [1] because our program is[0]. This can be confusing if we forget the first entry in an array is always 0. rb is an instruction to read the file in the byte (8 bits) rather than word (16 bits) size.

in, as has already been described, is a pointer to fname. Our computer's operating system attaches a unique character at the end of a text file. This character shows the End Of File (EOF) has been reached, so stop reading. This simplifies reading in the file since all we have to do is to set up a while loop to terminate on the EOF.

With the text saved in the array we are in a position to read it back. This is done by calling another function,

```
disp_txt(file_in,i)
```

We recognize file_in as the array name. The [] is not needed, in fact, if used it will result in an error message. The variable i contains the element count.

New names are assigned in the function call but they mean the same thing. The while begins with the final count and decrements with each pass until the entire text has been written out to the screen. On each pass putc writes a single character. If the char is '\r' putchar sends a tilde to the screen.

The insertion is done without disturbing the array counter. We see this in function disp_txt(text_in,j):

```
while(j--)      {
 putc(text_in[i],stdout);
  if(text_in[i+1] == '\r')
  /*  add end of line signal  */
     putch('~');
       i += 1;     }
```

The if statement looks ahead one character in search of a carriage return ('\r') showing the end of the line. When a '\r' is detected a tilde, '~', is written to the screen. The tilde will appear, we'll see it when running the re-compiled program, at the end of each line.

This is how ZANADU appears with the tilde.

```
In Xanadu did Kubla Khan~
A stately pleasure dome decree:~
Where Alph, the sacred river, ran~
Through caverns measureless to man~
Down to a sunless sea.~
So twice five miles of fertile ground~
With wall and towers were girdled round.~
A savage place! as holy and enchanted~
As e'er beneath a waning moon was haunted~
By woman wailing for her demon-lover!~
~
```

If we so wished the tilde could have been inserted in the text file and returned to the disk. That is an exercise you might wish to experiment with for your own benefit.

Array Summary

In this segment we have learned the basics of the array as used with the C language. We have looked in some detail at single index integer and character arrays with illustrative examples. We completed our study with the analysis of a program capable of reading a text file from disk into an array and then writing it back to the screen. We learned that if needed, revisions can be made to the material before or during its writing out to the screen. ■

This series continues with structures and unions in the next issue.

The Z-System Corner

by Jay Sage

Jay Sage has been an avid ZCPR proponent since version 1, and when Echelon announced its plan to set up a network of remote access computer systems to support ZCPR3, Jay volunteered immediately. He has been running Z-Node #3 for nearly five years and can be reached there electronically at 617-965-7259 (MABOS on PC-Pursuit). He can also be reached on Genie (as JAY.SAGE), in person at 617-965-3552 (until 11:30 pm), or by mail to 1435 Centre St., Newton, MA 02159.

Jay is best known for his ARUNZ alias processor, the ZFILER file maintenance shell, and the latest versions 3.3 and 3.4 of ZCPR. He has also played an important role in the architectural design of a number of other programs, including BGii, NZ-COM, and Z3PLUS.

In real life, Jay is a physicist at MIT, where he tries to invent devices and circuits that use analog *computation to solve problems in signal, image, and information processing.*

True, February is a short month, but somehow I don't think that explains why the **beginning** of February (the due date for this column) came quicker than usual. It is probably a good thing that I have convinced several others to start contributing regular columns to TCJ. That way, Art Carlson may be busy enough not to notice that the deadline passed without my column. Because it is late and because we now have quite a few prolific writers joining the TCJ ranks, I'm going to keep my column shorter than usual this time (I know I have said that before, but this time I think it really will be true).

For this issue I will be catching up on some correspondence, informing you of the imminent release of the first high level language for Z-System programming, bringing you up to date on Z-Node developments (there is a new Z-Node Central), and beginning a discussion of issues related to bringing up a remote access system (so that more of you might decide to set up a Z-Node).

Letters

Despite our requests for letters with your comments and suggestions, we don't get very many. Recently, however, Art forwarded to me two lengthy and thoughtful letters from James Ott. I would like to begin by addressing some of his comments and questions.

The ZCPR33 Programmer's Reference Manual

First, Mr. Ott asked about the *Programmer's Reference Manual* to which I made reference in my *ZCPR33 User's Guide*. Well, the truth is that after writing the user's guide, I really didn't have energy left for another major manual. Instead, I started to release programming notes one at a time as files on the Z-Nodes. Even at that, I only got to three of them. The files have names of the form Z33PNOTE.###, where "###" is a sequence number. Note 001 deals with the command status flag in the message buffer and the extensions in its use introduced with ZCPR33. Note 002 discusses proper coding techniques for shells under ZCPR33 and later. The third note covers the parsing of files by the Z33 and Z34 command processors. I am tempted to reproduce some of that material here, but then I would surely fail in my resolve to keep this column to a reasonable length. So you will just have to look for them at the Z-Node in your neighborhood. If it is not there, see if the sysop will get a copy from Z-Node #3.

Shells and ZEX

Mr. Ott continued: "The User's Guide mentions an addition to shell coding necessary to ensure the shell pushes its name onto the shell stack when it is called by ZEX." I think there is some misunderstanding here. It was under previous versions of ZCPR3 that special code was required in shells to deal with ZEX (and it never had to do with pushing anything onto the shell stack). Under Z33 and later, this code can (and, to make the programs smaller, should) be removed. It has now been quite some time since the release of ZCPR33, and I think that most shells are now coded in the more efficient way.

As we have discussed in previous columns, a shell command comes into play when the command line buffer becomes empty. In that case, if there is a command on the shell stack, the command processor invokes that command instead of asking the user for the next command line. In this way, the shell takes over the function of the command processor.

ZEX's function is similar to that of SUBMIT and XSUB together. While SUBMIT stores its script data in a disk file, ZEX keeps it in memory. This enables ZEX to run much faster, but it reduces the memory available to programs. A simple SUBMIT script feeds a series of commands to the command processor, thus doing under CP/M what the multiple command line of ZCPR does. SUBMIT can be useful even under ZCPR, however, because it can supply a longer string of commands than could fit into the command line buffer.

XSUB running under SUBMIT allows characters in the script to be fed not only to the command processor but also to programs as they run. This is what is called input/output (I/O) redirection. In this case it is **input** redirection; the operating system is made to take its characters not from its normal source—the keyboard—but from a disk file (SUBMIT) or a memory buffer (ZEX). This is both extraordinarily useful and extraordinarily tricky. Long ago I promised to discuss this subject at length in this column, but I have never gotten around to it. Bridger Mitchell, Joe Wright, and I are now engaged in a joint project to perform a major upgrading of ZEX, and I am sure it will be the subject of TCJ columns by one or more of us.

The ZCPR33 command processor observes the following hierarchy for acquiring new commands, where step 1 has the highest priority and step 5 the lowest. The way this hierarchy functions is described in more detail in the *ZCPR33 User's Guide*.

1. Commands pending in the multiple command line buffer
2. Commands from a ZEX script

3. Commands from a SUBMIT script
4. A shell command
5. User input

Under ZCPR30, ZEX did not appear explicitly in this hierarchy; it came into play only by virtue of its ability to redirect input at step 5. This posed a serious problem when a ZEX command was issued under a shell. Although the user intended the script to be performed as commands, the shell would take it as input to the shell instead. To avoid this, rather lengthy code had to be included in every shell to see if ZEX was running and, if so, to feed its input directly to the multiple command line buffer. This resulted in completely useless loads of the shell code for each line of the ZEX script. Execution was so slow and annoying that for all practical purposes ZEX scripts could not be run under shells.

The ZCPR33 command processor was redesigned to deal with ZEX explicitly just as ZCPR30 always did with SUBMIT. ZEX was placed above SUBMIT in the hierarchy so that ZEX scripts would function like arbitrarily long command lines and could be invoked from SUBMIT scripts.

The New Libraries

Mr. Ott had several questions about the assembly-language libraries that support the Z-System. I will not comment extensively here, but I would like to announce that new versions of the libraries, which have been in use by a number of us for many months already, will soon be made available to the public. I had expected them to be a commercial product, but, for several reasons, it now appears that the code will be made available at no charge. Only the manual, which will be quite a large book, will be sold. This is good news.

One of Mr. Ott's suggestions was that the Z3LIB routine called PRGLOAD, which is used for chaining from one program to another, be updated to allow for type-3 programs. This probably could be done without a great deal of difficulty, but I am not sure that it is worth the trouble. What about type-4 programs? Loading them is much more complex because of the need to compute the relocation to a run-time address. It seems to me that a better way for programs to chain to other programs is via the multiple command line buffer. If anyone can suggest any advantages of a direct load, I would be interested in hearing them.

The Command Line Tail

One of the mistakes in ZCPR30 was its failure to check for command line tails that would not fit in the buffer from 80H to FFH. CP/M had no problem with this because the whole command line was not long enough to allow this to occur. With a 200-or-more-character command line buffer, there is nothing to stop a user from entering a command with a tail longer than 128 bytes. Under Z30 this caused a catastrophe, because the tail was copied into the buffer **after** the program was loaded, and then the tail could overwrite the beginning of the code. Z33 and later monitor the length of the tail and stop copying before this can happen.

With type-3 programs that load at addresses higher than 100H, longer tails could be copied without damaging the code, and Mr. Ott requested that this be implemented in future versions of the command processor. I think this would be a very bad idea. One of the reasons for using type-3 programs is so that any code residing at 100H can be run again later using the GO command. If the type-3 program's tail overwrote the TPA, then trouble would occur when the GO command was executed.

Moreover, there is absolutely no need for such a feature. If a program wants to support a tail longer than 126 bytes (it doesn't even have to be a type-3 program), it can simply read its arguments not from the buffer at 80H but directly from the multiple command line buffer. As an aside, I have thought of making the command processor not convert the command line buffer to upper case. Then programs could process lower case input directly (as in MS-DOS) without the special symbols to indicate case shifts as in the ECHO and IF INPUT commands. The command tail buffer at 80H must be converted to upper case for compatibility with CP/M programs, which assume that that will be done. Besides the fact that there might be a significant code penalty (the command tail buffer and the command file control block would both have to be individually case shifted), I worry that there may be a number of Z-System programs that either rely on the command line being in upper case or, worse, convert it to upper case. I'd be interested in hearing any opinions on this topic.

Still More on Z3 vs. Z2 Shells

Mr. Ott sent me a lengthy letter with some interesting examples of the use of the shell stack. I think his is the first response I have received that pointed out an aspect of the shell stack that I had not previously appreciated.

I don't think about shell command lines with more than one command, so the following distinction never occurred to me. One can think of the shell stack as containing not just, say, four commands but rather four **groups** of commands. The shell stack not only holds these commands; it also provides the mechanism for grouping them, for providing parentheses, if you will. The Z2 approach to shelling would have a very hard time doing this. For example, if the first element on the shell stack contains the command line "CMD1A;CMD1B" and the next lower element the line "CMD2A;CMD2B;CMD2C", the equivalent Z2 situation would have the command line

```
CMD1A;CMD1B;CMD2A;CMD2B;CMD2C
```

The SHCTRL POP command removes an entire Z3 shell entry, containing possible multiple commands. How could we implement this function with a Z2-style shell? Even if each command were marked with a "/s" token, one could still not tell which ones were grouped with which. There might be a solution to this problem, but the Z3 shell approach certainly handles it nicely.

Mr. Ott's letter included a very interesting application example. I've made a few changes that, I hope, do not harm its essence. When the computer is booted, the STARTUP alias loads the command sequence "FIRSTSH;MENU", where FIRSTSH is a special utility that places the following command sequence onto the shell stack:

```
PARK;VID RST.MSG;STOP
```

When this sequence runs, it will park the heads on the disk drive, display a message to shut off the computer, and run a program called STOP that locks the machine. Of course, this multiple-command shell could easily be replaced by an alias SHUTDOWN.

This termination shell does not run immediately because of the command MENU following it in the command sequence. This loads a second shell onto the shell stack. The user then lives inside the MENU shell for some time. At some point, the user may make a selection that generates the command "CD WORK:". This would log into the WORK: directory and issue an automatic ST command there. The ST alias might have the script

```
LDR WORK.NDR;SHCTRL POP;ZFILER
```

This would install a new set of named directories, pop MENU off the shell stack, and replace it with the ZFILER shell. The user could then do some work using ZFILER. From ZFILER, a macro command might make another shell change by popping the shell stack and installing another shell.

The above process would continue until the user made an exit selection. This would pop the shell stack without installing a new shell. As a result, that first line we put on the shell stack would become active, forcing the computer to be shut down in an orderly, preplanned way. Of course, this approach is not totally foolproof if the user has access to the power switch or power cord. But it certainly helps.

Mr. Ott worried that this example could not be achieved using a Z2 shell system. I think it could, though not with quite the same ease and elegance. The STARTUP alias would contain the line

```
MENU;SHUTDOWN
```

When MENU ran, it would substitute for itself in the command line the desired command line plus its own reinvocation command with the flag "/S" to mark it as a shell. Thus the command line would become

```
USERCMD;MENU /S;SHUTDOWN
```

For the user command CD WORK:, the command buffer would evolve as follows:

```
CD WORK:;MENU /S;SHUTDOWN
ST;MENU /S;SHUTDOWN
LDR WORK.NDR;SHCTRL POP;ZFILER;MENU /S;SHUTDOWN
```

When the SHCTRL command ran, it would scan the command line for the next shell command as marked by the "/S" flag and remove it from the command line. This would leave one with

```
ZFILER;SHUTDOWN
```

When ZFILER generates a macro command, the command buffer would have

```
MACROCMD;ZFILER /S;SHUTDOWN
```

This would continue until one canceled the shell. Then the SHUTDOWN alias would run.

Mr. Ott was concerned that the command line would grow longer and longer as old shell commands piled up. Indeed, this would happen if there were no 'popping' mechanism. The "/S" marker I proposed in my previous columns is critical here, since without it there would be no way to tell which command to pop. This approach, I think, would fail with aliases as shell commands. The Z3-type shell really helps us in this case.

High-Level-Language Support for Z-System

I have some especially exciting news about the first high level language specially designed for Z-System. In late January, I got a phone call from Leor Zolman, author of BDS C. He was interested in bringing out a new version of his C compiler for the modern 8-bit market. After some consultation with me on what was needed to support Z-System, Leor went to work on the run-time code, and in less than two weeks he had a version ready for beta testing.

He came over to my house to demonstrate the result, and I was amazed to see how easily one could write a utility, complete with named-directory support, even including password protection. A little fine-tuning is still needed, but I am already excited about the impact that the availability of this quality high level language will have on Z-System development. Marketing details have not been worked out at this point, but you can expect the new BDS C to be available at an attractive price through Z Systems Associates (ZSA) channels (Plu*Perfect Systems, Sage Microsystems East, Z-Nodes, and Z-Plan user groups).

The New Z-Node Central

Ron Bardarson, who had operated Z-Node Central after David McCord retired as the sysop, decided to change the focus of his system and his node designation from Z-Node to X-Node. This was to indicate its experimental nature and its focus on software and systems development rather than on the distribution of Z-System software. Its number remains 408-432-0821 (CASJO on PC-Pursuit).

To maintain a center of support for Z-System users and for Z-Node sysops, we have established a new Z-Node Central. Richard Jacobson, whose Lillipute Z-Node in Chicago has long been, in my opinion, the premier Z-Node in the country, has agreed to take on this new role. He will continue to use the delightfully ironic Gulliverian name (at over 100 Mbytes, his is the least Lilliputian of the Z-Nodes), but his node number now drops from 15 to 1.

This new function is added to several special services that Lillipute already provides. It is the official bulletin board and remote access system for both the North American One Eighty Group (NAOG) and TCJ. The full system, comprising two independent computers, is available on a subscription basis. All users who provide registration information will get 15 minutes per day of free access. Unlimited access to both computers (within reason) is available at a rate of $25 for six months or $40 for a full year. TCJ subscribers and NAOG members get free access to limited areas and can upgrade to full subscriber privileges at $5 less than the standard rates. Z-Node sysops will have free access to the full system. The phone numbers are 312-649-1730 and 312-664-1730, accessible via the ILCHI outdial of PC-Pursuit.

System Security Under Z-System

As part of my plan to build up the network of Z-Nodes, I would like to begin a series on issues related to setting up a remote access system (RAS) using Z-System. Many people do not realize it, but NZCOM and Z3PLUS (the automatic Z-Systems for CP/M-2 and CP/M-Plus, respectively) create systems with the full security capability necessary for a RAS. It just takes a few simple operations to engage it.

These issues have been brought to my attention recently because I have been in the process of setting up a second remote access system at my house. This one is for the BOSKUG group of the Boston Computer Society. It will be a two-line system running on a very powerful 16 MHz Kaypro 286 computer with multitasking software. For all this power, however, I was struck by the tremendous complexity of setting up a remote system on such a computer, all because of the lack of a secure operating system.

With MS-DOS, a remote system absolutely cannot allow a caller to gain direct access to the operating system command prompt, because once he has that access, there is no way to limit what he can do. It made me realize how fortunate we are with Z-System to have an operating system with enough security to permit callers on a remote system to run the system more or less as if they were sitting at the keyboard of their personal machine. They don't have to have an elaborate apparatus standing between them and the system.

There are two main aspects of that security. One is the wheel byte. This system flag is tested by many Z-System programs to determine whether certain operations should be permitted or denied. Commands for doing things like erasing, renaming, or copying files typically require that wheel status be in force. Other commands will allow some operations to non-wheel users but deny other operations. For example, some directory programs allow writing an image of the directory to disk or to the printer. These options are (or should be) restricted to 'wheel' users. The wheel byte itself is set and cleared by special commands, such as the WHL command of the RCP. Obviously, a password must be entered correctly before WHL will set the wheel byte.

The second and more complex security feature in Z-System concerns the facilities for limiting the disk areas which a user can access. Many users are unaware of these features, and even those who are aware of them often do not understand them fully and clearly. I will cover the major points here.

With version 3.3 of ZCPR, I introduced extensive and significant changes in the way directory references and security are handled. These changes made understanding security more complex for the system implementor but much easier and less intrusive for the user.

ZCPR3 supports two basic forms of directory reference, the disk/user or DU form and the named directory or DIR form. We will assume that the reader is already somewhat familiar with the basic concepts. The DU form is native to CP/M, which knows about disk drives from 'A' to 'P' and user numbers from 0 to 31 (though there are restrictions on user numbers above 15). The DU

SAGE MICROSYSTEMS EAST

Selling & Supporting the Best in 8-Bit Software

- New Automatic, Dynamic, Universal Z-Systems
  - Z3PLUS: Z-System for CP/M-Plus computers ($69.95)
  - NZ-COM: Z-System for CP/M-2.2 computers ($69.95)
  - ZCPR34 Source Code: if you need to customize ($49.95)
- Plu*Perfect Systems
  - Backgrounder II: switch between two or three running tasks under CP/M-2.2 ($75)
  - ZDOS: state-of-the-art DOS with date stamping and much more ($75, $60 for ZRDOS owners)
  - DosDisk: Use DOS-format disks in CP/M machines, supports subdirectories, maintains date stamps ($30 - $45 depending on version)
- BDS C — Special Z-System Version ($90)
- SLR Systems (The Ultimate Assembly Language Tools)
  - Assembler Mnemonics: Zilog (Z80ASM, Z80ASM+), Hitachi (SLR180, SLR180+), Intel (SLRMAC, SLRMAC+)
  - Linkers: SLRNK, SLRNK+
  - TPA-Based: $49.95; Virtual-Memory: $195.00
- NightOwl Software MEX-Plus ($60)

Same-day shipping of most products with modem download and support available. Order by phone, mail, or modem. Shipping and handling $4 per order (USA). Check, VISA, or MasterCard. Specify exact disk format.

Sage Microsystems East
1435 Centre St., Newton Centre, MA 02159-2469
Voice: 617-965-3552 (9:00am - 11:30pm)
Modem: 617-965-7259 (password = DDT)(MABOS on PC-Pursuit)

form of directory reference is basically physical in nature. Drive letters are associated with real physical devices, and the files in all user areas associated with a given drive letter are stored on the same physical device. One can think of this directory structure as spanning a flat, two-dimensional space (in contrast to the hierarchical tree-structured directories of Unix or MS-DOS).

While the DU directories are basically physical, named directories are purely logical constructs. The named directory register (NDR) module in a Z-System contains a mapping of directory names to drive/user values. The user can load different sets of directory associations at different times. Thus, unlike the static (fixed in time) directory structure of Unix and MS-DOS, the directory structure of Z-System can be dynamic (changing in time).

When the DIR form is used, the command processor or a Z-System program looks for the name in the NDR and substitutes the drive and user values. Only drive/user values are used in the actual file references processed by the disk operating system (DOS). Named directories provide two different and important functions. One is convenience. It is much easier to remember that one's assembly language tools are in ASM and one's wordprocessing files in TEXT than it is to remember that the directories are A7 and B13. The second purpose of named directories is to provide security.

Access to directory areas in a Z-System is controlled in both the DU and DIR domains. Under Z30 these two control mechanisms were completely independent; under Z33 and later, as we shall see, they are very closely coupled. The limits in the DU domain are set by values in the environment descriptor (ENV) called max-drive and max-user. They define a rectangular area of allowed directories in the flat directory space, with drive values ranging from 'A' to the drive specified by max-drive and user numbers ranging from 0 to the number specified by max-user. The smallest space still includes the boot directory A0.

Named directories offer a more flexible means of controlling access to areas on a system. The user can access a named directory even if it refers to a DU area that is beyond the bounds defined above. Each directory name can have an optional password associated with it. Whenever a reference is made to such a directory, the password is (should be) asked for and access granted only if it is entered correctly.

There are a number of important exceptions to the two security limits described above. One concerns the command search path specified in the Z-System PATH module. No restrictions whatever are imposed on the DU areas specified there. If the user was able to place a directory into the path, then it will always be scanned as necessary by the command processor, even if it would no longer be explicitly accessible.

Another general exception occurs for the standard (but optional) configuration of the command processor called WPASS. With this option, when the wheel byte is set, directory passwords are ignored. The user can then freely make reference to any directories either within the specified DU range or associated with a named directory (with or without a password).

Versions of the command processor since Z33 also make the assumption that if a user is in a given directory at the present time he must have had the authority to get there. Therefore, the command processor will always accept any reference to the currently logged directory, even if the DU is out of range and it has no unprotected directory name.

Another set of exceptions relates to the interplay of the DU and DIR limits. Recent versions of the command process act on the principle that if reference to a directory in one form (DU or DIR) would be allowed then references using the alternative form should be equally allowed. For example, suppose that the max DU limit is set to B3 and that directory C4 has the name DIRNAME with no password. Z30 would have refused to accept a reference to C4:, even though it would have no complaints about

(Continued on page 39)

Information Enginering

The Portable Information Age

by C. Thomas Hilton

There were a number of topics I wanted to cover in this issue, which will have to wait until another time. Borland has announced a new version of PARADOX. PARADOX3 has all the fine presentation graphics of QUATTRO, and an enhanced QUERY mode. As my deadline for this issue grows near, my copy of PARADOX3 has not arrived, and there is little sense in covering topics that will be soon obsolete.

We will discuss, however, a number of little "tricks" that will make your engineering of information a bit easier. But first, another thrilling tale of the adventures of "Fearless Leader, Project Manager."

The Portable Information Age, or, Ooooooops!

It was a slow Friday. The Fearless Leader was in another panic mode, preparing documents for an important presentation. I figured it was a good time to make myself scarce. I figured right.

The following Monday Fearless Leader was in the final throes of extreme displeasure, which had obviously lasted the weekend. It seems that, between his clerk and himself, something went amiss. In the middle of his best performance ever he noticed that most of the information he was discussing was not in the handouts distributed to all the dignitaries. Somewhere, somehow, his last minute changes did not get printed. After eating more than a little crow, and between embarrassed apologies, he was directed to an alien computer which contained an alien set of software and attempted a little "damage control."

Needless to say, Fearless Leader was in no mood to listen to my usual speeches about backups, data control, or proof reading documents. He had learned solemn truths in the worst way possible.

So, my friends, we will begin this issue's conceptual discussion about the Portable Information Age. In the discussion which followed "Black Monday," the events described above, it was thought that one would be protected if one assured that Fearless Leader had a pocket full of disks containing critical data. Well, that is fine, if you have a computer available things go wrong.

Well, when things go wrong, they seem to always go terribly wrong. Even with critical data with you, on disk, there is the problem of an alien computer, assuming it is compatible, to be dealt with. Next there is the problem of alien software, an alien printer with alien printer codes.

The obvious answer was to either provide a complete software environment, that could be booted in the alien machine without disturbing the existing environment, or make the information mobile.

It is important to note that it isn't enough to modem data to where it is needed, or FAX it, if only the remote person knows what is needed.

After developing software for all the above options, let it suffice to say that Fearless Leader now carries a TANDY laptop computer. Mine would arrive about a week later.

So, here we are. The most sophisticated information software in the world will seldom fit on a dual 720K floppy disk system. Most laptops are XT class machines. The majority of truly portable machines have LCD display screens that try their little chips out to emulate a color display. It will take some time to realize what problems this presents for the Information Engineer. It also takes some time to learn to appreciate the incredible value of being totally portable.

Information Engineers are behind the times when it comes to keeping up with information needs. When information is locked to a desk, apart from where it is needed, we are not doing our jobs. This is a sad, but true fact.

Also true is that, after years of depending upon sophisticated, high power, software we will be forced to look back to simpler times for solutions to modern problems. Once upon a time we were concerned with small file size and speed of program execution. Then came big and fancy development tools, and an easy life. In the future, do not be surprised if you find yourself digging through trunks and boxes for products like the BORLAND DATABASE TOOLBOX. With Turbo Pascal 5.0 and the DATABASE TOOLBOX a marginal programmer can create mobile information software in a few hours. These products have sure made me a "local hero" in the past few weeks. Good software for small, slow, mobile machines, which is attractive on LCD and gas plasma displays, will be in high demand. Prepare yourself.

"Good software for small, slow mobile machines, which is attractive on LCD and gas plasma displays, will be in high demand. Prepare yourself."

New Data From Old Files

The biggest problem in dealing with information is that you adopt the premise that there can never be too much information; that old information should be recovered. "You never know when it will come in handy."

Another real problem is the creativity and competence of data entry personnel. One would think that it would be a simple thing to enter something, as elementary as a name, in the same way, twice. Unfortunately, some think everything should be entered in upper case, some with only the first character capitalized, some with a comma after the last name, some feel no comma is needed. With this data entry creativity, a list of names, as a simple example, looks chaotic at best.

One solution is to force data entry into a common format. The problem is, someone will do their best to enter the data the way they feel it should be. In all cases productivity decreases. The person normally called upon to deal with the situation is the Information Engineer.

One solution I have adopted is to let the data entry people sate their desires, within reason. In this way, productivity is at a maximum. Each person enters the data in the way that is best suited to their personality. I am viewed as less a tyrant than I would normally be, and a great deal of aggravation is saved all around. With the data in hand, we can convert data fields to a consistent format.

Old data files are seldom in the format needed by current standards and policy.

Once again, I have selected PARADOX to deal with this task. Most database products have similar ways to accomplish the text formatting tasks we will discuss in this issue. I prefer PARADOX because the "language" is simple, straightforward, and easily understood by nonprogrammers.

LISTING ONE

```
PURPOSE:
  This script changes mixed upper and lower case entries to a
consistent first character upper case format for each word in the
field, ''JOHN JONES'' becomes ''John Jones''

USAGE:
 1. Place the cursor in the alphanumeric field to be converted.
 2. Press F10 S P and select Set_Char as the script to play
 3. Script may take a while if the table is a large one

EditKey                  ;Enter edit mode
Scan                     ;Loop, scan all records in table
  ExistingField = []     ;Move Current Field Into variable
  FirstVariable = Format(''CL'',ExistingField)
                         ;Make all characters lower case
  SecondVariable= format(''CC'',FirstVariable)
                         ;Now make first characters Caps
  [] = SecondVariable    ;Put the data back in the field
EndScan                  ;End of scan loop
Do_It!                   ;Exit Exit Mode

end of script
```

The first problem we will attack is the capitalization issue. For this example, the required standard is that the first character of each word in the data field is to be capitalized, the rest are to be lower case. The PARADOX script we will present is simple, and quite easy to understand. Its benefit is beyond value.

Listing 1 shows our solution to the capitalization problem. The "script" begins by placing PARADOX in the field editing mode. The script assumes you have placed the cursor in a text only field, such as a name field.

The SCAN statement forces PARADOX to look at every record in the table from the cursor position to end of file. It forms a "loop," in computer-speak.

Next we establish a variable to hold the current field. We will call this variable "ExistingField" because it will hold the data as it is found in the raw data record. The "[]" sequence tells PARADOX to deal only with the current field.

The next sequence is to format the data in the ExistingField variable, and place it into another variable. The PARADOX command FORMAT("CL",ExistingField) tells PARADOX to convert all text into lower case. FirstVariable then contains the text characters in all lower case.

The next FORMAT command is used to convert the first character of each word into upper case format, leaving it in SecondVariable. SecondVariable is then placed back in the current field.

The ENDSCAN statement forms the end of the loop, indicating that all commands between the SCAN statement and the ENDSCAN statement are to be executed for every field.

When all fields in the data table have been dealt with, we simulate the pressing of the "DO_IT!" key to assure we are not in the field editing mode when the script has done its work. This script will not work on numeric fields.

A quick look at your data table will show that all text characters have been formatted as desired! Such a small and relatively simple script contains great power, and is easy for most people to understand, with a little study. This is one of the many reasons I am so fond of PARADOX!

Converting Number Formats From Old Files

When looking for a blank disk, which is rare at the "Hermit's Cave," I found some test data that, were it in proper format, would save a great deal of data entry time. The problem is that the decimal data was entered in integer format. The graphics routines used in the old application could deal only with integer data.

It was decided to translate the old data file into a format PARADOX could understand, and do the conversion there. The need here was essentially divide any number greater than ten, by ten, thus rendering a decimal representation of the data. PARADOX is very forgiving and flexible, when dealing with numeric formats. In fact, without specific programming, it has only one common numeric format.

Listing 2 shows a simple script written to make the conversion. You will note that it is similar to the previous example, and operates in the same fashion. Place the cursor in the field to be converted, and "play" the script.

LISTING TWO

```
PURPOSE:
  This script searches each record in a table's column looking
for any value that may be larger than 10.0. A field value greater
than 10.0 needs to be converted into its proper decimal form for
use in the Paradox system.

USE:
1. Access a table structure in the normal way, using the VIEW mode.
2. Place the cursor in a table column with an offending value.
3. Press F10 S P and select TO_DEC as the script to play
4. Upon exit table column will have values converted to
     decimal format

EditKey              ;Enter Edit Mode
Scan                 ;Enter scan loop for all records
  WorkVariable = []  ;Move field into variable
                     ;If the field has a value greater than 10
                     ;then it needs to be corrected by shifting
                     ;decimal place to the right one place by
                     ;dividing the value by ten
                     ;
  IF WorkVariable >10 then WorkVariable = WorkVariable /10
    [] = WorkVariable    ;Move the value back into the table field
  EndIf              ;End of conditional IF  statement
EndScan              ;End of scan loop
Do_It!               ;Exit edit mode

end of script
```

If the previous example has been understood, Listing 2 should present few comprehension problems. You may note that it is a simple variation on the previous theme. Once a concept is understood, and assumed valid, minor variations can produce powerful results with the PARADOX system.

Preparing Data for Conversion to Other Formats

When I have to do some serious Information Engineering, I like to move data from one format to another. Normally I want to move the data into PARADOX for processing. Many products can read, and write, dBaseIII file formats. This is the format I normally use to move data between various products and PARADOX.

For mobile information systems, particularly those I write in Pascal, a different intermediate format is required. The easiest to deal with is what is called the "Comma Delimited Format." This is the native format used by many BASIC languages. Essentially, the data fields are represented by ASCII characters in a single line of text. The end of line, represented by a carriage return, or carriage return and line feed sequence, represents the end of the record. A string data field is surrounded by quotes. Numeric fields are represented by a series of numeric characters. Quotes are not used unless the numeric value is of a string data type. In all cases the field is delimited, or separated from other fields, by a comma.

The comma delimited data format is very easy to deal with in a text editor. What I normally do is load the data structure into SPRINT, and replace each comma with a carriage return using the "search and replace" function. With that accomplished, I remove all the quotation marks with the same process. This all works fine, unless there is a comma in a string field, as you might find separating the first and last name.

If a comma is used by the data entry person, then the number of fields per record will not be left constant during the initial formatting. Remember that I blindly asked SPRINT to replace each comma with a carriage return so that each field occupied an individual line. With each field on a line by itself data can be read into a program in a "batch mode," simulating keyboard input. Now then, if there is a comma in a string field, one data record will end up with more lines per record than another field, upsetting the input sequence. For this process to work each data record must have the same number of field-lines. An unexpected comma splits the name field, for example, into two separate fields, adding what will later appear to be an extra field in a data record.

While it would be just as easy to write a program to accomplish this conversion of the data, it would still see the comma in a string field as a field separator. A word processor does the job just as well. The only way to make this process work is to remove all of the commas from the string fields in the data table.

LISTING THREE

```
PURPOSE:
 This script removes commas from the current data field to
prevent confusion by information systems reading comma delimited
data formats.

USE:
1. Access a table structure in the normal way, using the VIEW mode.
2. Place the cursor in a table column with an offending value.
3. Press F10 S P and select TO_DEC as the script to play
4. Upon exit table column will have values converted to
     decimal format

EditKey                  ;simulate the pressing of F9
SCAN                     ;form the scan loop indicator
  X= MATCH([],''..,..'',FIRST,REST)
                         ;try to parse the field
  If X Then              ;if there was a comma found
    []= FIRST + REST     ;recombine the parts
  ENDIF                  ;end of IF statement
ENDSCAN                  ;loop until done
DO_IT!                   ;leave edit mode

end of script
```

Once again, the short and simple PARADOX script shown in Listing 3 does the job for us. The mechanics of the script are the same as the other scripts used in this article, as examples, with one exception. The MATCH statement.

The MATCH statement works the way most of us have always wanted a string parser to work. No matter what the language, there is a command to locate the position of a given character, or string, in another string. This location is returned as a numerical value. It is then left up to us to parse out the characters before the location, or Location -1, and the remainder of the string from Location $+1$. This avoids including an offensive character, such as our comma. The PARADOX MATCH statement does all of these tasks for us.

MATCH is a function, it returns a numerical value. In this application of the function, we do not care what the position of the comma in the string is. The "x" variable is therefore a "dummy variable," it just keeps the function happy.

We tell MATCH to look in the current field ("[]"), for the occurrence of a comma, ignoring all that comes before and after, ("..,.."), where the double dots form the "don't care" indicators. If there is a comma found in the current field, "x" will be a nonzero value, indicating the position in the string of where the match was found. Again, we don't care where the comma may be found, only that there may be one.

If a comma is found, the conditional "IF" block will be executed, otherwise it will be ignored. If a comma was found, the string will be parsed into two parts. The first part will contain all characters up to, but not including the comma, the rest of the string, also excluding the comma, will be placed in the second string. These two strings are represented, in Listing three, by the variables FIRST and REST. The two parts of the original field are then combined and put back into the current field. The process then repeats for each field record in the data table.

This is a very important, and an otherwise difficult process to accomplish. PARADOX makes it easy with a simple script.

Preparing Statistical Data

Since this section is considered suitable for "advanced" users, you may feel that it may be more complicated, harder to understand. Well, you would be wrong. The more difficult the task, the easier PARADOX makes it to deal with.

From the QUERY mode we can ask for the average of a group of numbers, such as test scores, and get an answer. This is as simple as entering, "CALC AVERAGE," into the query fields desired. A more detailed analysis of our data is slightly more difficult.

LISTING FOUR

```
PURPOSE:
 This script calculates the standard deviation and a sample
deviation from the average of a numerical field.

 USAGE:
 To use this listing on your data table, you must substitute
''TABLENAME'' with the name of your data table (in quotes), and
''FIELDNAME'' with the name of the data field to be processed.

PRINTER ON               ;Send output to printer
+ (See Text)
? ''The Standard Deviation For MD+ Is
   '',CSTD(''TABLENAME'',''FIELDNAME'')
? ''The SAMPLE Standard Deviation Is'',
  SQRT(CVAR(''TABLENAME,''FIELDNAME'')*
  (CCOUNT(''TABLENAME'',''FIELDNAME'')/
  (CCOUNT(''TABLENAME'',''FIELDNAME'')-1)))
+ (See Text)
PRINTER OFF              ;Turn Off Printer Echo

end of script
```

For some manipulations of our data we can only use scripts. It is for this reason that PARADOX scripts have been presented in this issue. In addition to the average values of test score data, the client wants to know what the Standard Deviation and Sample Deviation of the scores are.

(Continued on page 39)

Computer Aided Publishing

By Art Carlson

What is Publishing?

Publishing is one of the business areas where computers can be extremely useful. In fact, it's almost impossible to do business without them. That's one of the reasons why you hear so much about DTP (Desk Top Publishing), and why it is one of the few growth areas available for micros right now. But, there is a lot of hype and confusion about what publishing really is.

The purpose of this series is to clarify what is involved in publishing, how micros are being used in publishing, and to envision what can be done in the near future. We will cover wordprocessors, page preparation, graphics, laser and high definition printers, and data management. We'll also cover font generation and loading, plus PostScript and H-P PCL programming and setting type directly from database and other programs. It will be necessary to include information on typography and the printing process in order to understand how what we do relates to the final product.

Publishing involves a lot more than just generating camera-ready copy. I break it down into the four broad categories of Marketing, Manuscript, Production, and Fulfillment. Marketing (which you'll notice I placed first) is of primary importance. Before you decide to publish a manuscript, you have to determine the need, who wants it, will they buy it, how much will they pay, what should it look like (what size, hard bound or paperback, plain or lots of color, etc.) how you will solicit their order, etc. After it is produced, you'll have to promote and sell it.

I placed the manuscript second, even though most people complete the manuscript without any thought of determining its marketability. This is unfortunate because quite often the manuscript must be completely redone in order to match it to its market. Even worse, perhaps it shouldn't be published at all! At any rate, at this point, after much research, editing, rewriting, proofreading, and more rewriting, you feel that you have a manuscript ready to go. You did do some copyfitting to be sure of the final page count and to make sure that it fits on an integer number of press signatures, didn't you? By the time the manuscript is finished you should be very familiar with your wordprocessor.

With the completed manuscript in hand, you start thinking about producing it. That's unfortunate, because marketing, manuscript preparation, and production are mutually dependent, and production decisions should have been made in parallel with the other activities. At any rate, now you decide exactly what the trimmed page size will be, the size of the active text area on the page, and how it is going to be printed and bound. You design the piece, including front and back matter, table of contents, index, etc.

With all the facts in hand (or at least in mind), you set up a style sheet and pour the text into your DTP program (right now I'm using PageMaker 3.0 on an AT clone). Then you either dump it out to a laser printer (this is set on a H-P LaserJet II), or you send it out to a service bureau for high resolution output.

Finally, you send your labor of love to your chosen printer/binder with lots of $$$ and wait for the trucking company to deliver the pallets of books. While you are waiting, you'd better be promoting and lining up sales so that you can pay the bills.

After the orders start pouring in, you have to maintain records for the tax authorities, generate shipping labels and invoices, and keep track of your customers because they are your best prospects for your next book. You also have to pack the books and deliver them for shipping. The marketing and fulfillment, including answering phone calls and letters, can be so time consuming that you never have time for your second book!

How Important is DTP?

DTP is not really publishing, it's just the generation of camera ready copy. And that is a very small portion of the total publishing effort. I estimate that I only spend about 5% of my time on the DTP portion.

Publishing is the business of planning and overseeing the marketing and production of printed matter (vieos and CDs are another matter which I'll ignore here). You don't even have to use a single computer in order to be a publisher. Everything could be jobbed out to service bureaus and freelancers

Jeffery R. Parnau, in his book *Desktop Publishing: The Awful Truth*, states that it should be called Tabletop Typesetting instead of DTP. If you are thinking about DTP, or even just want to know what it is really about, get his book (the address is at the end of this article). It could save you thousands of dollars and many sleepless nights. This is the first book I've seen which really tells it like it is--all the others are just trying to sell you products which may not work!

I use computers for three primary purposes in the publishing business: 1) Wordprocessing, 2) Database and information management, 3) Page preparation. I could continue to operate without the database software by reverting to manual systems, and I could prepare pages by pasting up galleys, but I could not continue without a wordprocessor because I want to write. If someone left me with DTP and database software, but took away my wordprocessor, I'd quit and do something else!

DTP is a wonderous development which will greatly facilitate publishing by providing Tabletop Typesetting. But, it is page preparation and not publishing. I've gone overboard to prove a point, and I've stacked the deck by talking about books. Someone who needs to produce a dozen posters about the company picnic will have

(Continued on page 40)

Shells
ZEX and Hard Disk Backups
by Rick Charnes

I'd like to start off this issue with an appeal to all users of ZCPR3 without a modem to give serious consideration to getting involved in the BBS world. The nationwide network of Z-Nodes we have established is perhaps the best way we have to educate ourselves in the many aspects of our extraordinary operating system. In the long intervals between TCJ issues this network remains our foremost means of communicating with each other. Its sense of immediate action and response with many individuals helping each other and sharing programs that we have written is perhaps the single most important source of the energy behind the Z world. All the ZCPR3 columnists you read here in TCJ—Jay Sage, Bridger Mitchell, Bruce Morgen, Cam Cotrill, Hal Bower and myself—are all active participants in this network. I cannot emphasize how much this kind of sharing and give-and-take can add to your enjoyment of computing. And it can all be done easily with a minimum of phone charges via PC-Pursuit.

The PC Pursuit network allows those with a modem living in the metropolitan areas of the U.S. to access any bulletin board system nationwide (again, in the metropolitan areas of the country) during weekends and non-business hours for a flat monthly fee of $30. This makes it easy and inexpensive to call, among many others, two of the most popular Z-Nodes, the official TCJ BBS at (312) 649-1730 and Jay Sage's Newton Centre Z-Node at (617) 965-7259. For more information about PC-Pursuit call Telenet at (800) 336-0437.

There has been much discussion on the Z-Nodes in recent months on the subject of ZEX, the ZCPR3 world's preeminent 'batch processor'. It is partly occasioned by Joe Wright's release in autumn of 1988 of ZEX version 4.03, partly from continued enthusiasm over Jay Sage's release of NZEX-D quite some time ago and his incorporation of ZEX into ZFILER's magnificent group macro facility, and partly due to the fact that it is simply in general an extraordinary program allowing for tremendous opportunities for automation where nothing (not even King ARUNZ!) else will do. If I am not mistaken, Bridger Mitchell's column this issue is devoted to a reconceptualization of what ZEX should be, and as a fervent admirer of the program (I use it wherever I can!) I am very much looking forward to further developments of this program by Bridger and his colleagues.

In that spirit, then, and as mentioned last time, I would like to attempt to give readers a flavor of what ZEX can do in the context of discussing one particular use to which I put it, the BACKUP.ZEX script I use for backing up my hard disk directories. If others have feared ZEX because of its reputation for eccentricity and exotic nature, it is perhaps for this very reason that I have embraced it. ZEX might be compared to the CP/M SUBMIT the way one might compare the experience of listening to a Top 40 radio station to attending a live performance of the 1812 Overture given by the San Francisco Symphony. Hopefully between my and Bridger's column you will be inspired to archive good old SUBMIT onto a sturdy but yellowing floppy and move on up to the ethereal realms of ZEX.

This column is intended generally for those with some prior experience with ZEX. I am not here going to explore all its nooks, crannies and meadows, filled as they are with fragrant and variegated bloom, though that would certainly be a wonderful and worthwhile venture. It will not be a primer on the basics of its use. I will of necessity touch on some of its features, but my purpose here is mostly to explore a few particular examples of how in my BACKUP.ZEX script I have interfaced ZEX with both the ZCPR3 shell variable subsystem and the ZCPR3 registers, the combination of all three of which is guaranteed to be a pick-me-up to the most tired of "there's-nothing-left-to- explore-in-CP/M" ne'er-do-well's. In studying some of these fairly specific ZCPR3 and ZEX programmatic techniques I am taking a gamble and hoping that the reader can draw some general lessons.

Like a goodly number of hard disk owners I have spent many (generally happy) hours trying to find the absolutely perfect program to perform my regular periodic backup of those hundreds of precious files I have lovingly created over the years. Sadly I have found no such program. I have yet to find any scheme that addresses what I believe to be the central dilemma of a hard disk backup program: how to ensure during one's periodic backups on hard disk directories requiring more than one backup floppy, that with each periodic backup any given file is copied onto the same backup floppy.

When backup programs that have no provision for this are run for the first time, they simply fill up, in completely random order, a given number of floppies with the files from the hard disk. Each successive run copies the files in the same or another random order.

Suppose, however, with this first run of the backup program my ALIAS.CMD happens to get copied onto the third floppy in a set of five from my root directory. Then, next weekend when I run the backup program again, assuming I've modified ALIAS.CMD and the archive bit is therefore turned off, it will again be copied by the program. But there's nothing to guarantee it will this time be copied to the same floppy! It may end up on the first, or second, or fifth, disk. After several runs of the backup program I may end up with several different copies of ALIAS.CMD on different backup floppies, modified at different times on the hard disk! I have yet to see a satisfactory solution to this problem in terms of a single program.

In the true spirit of "Z," therefore, or what Frank Gaude' used to call "the tool approach," I have written a ZEX file to do the job. I have decided the best solution lies in copying the files onto the floppies in alphabetical order. I have settled on six for the number of backup floppies required by each of my major hard disk directories. I backup onto floppy one all files beginning with the letters A through D, floppy two gets the files beginning with E through J, and so on through the alphabet, reserving floppy six for the "Z" files of which for some mysterious reason there always seem to be an inordinate number on virtually all my directories...

I do this with 26 separate invocations of PPIP (renamed to COPY), one for each letter of the alphabet, used with its '/ae' op-

tion to indicate "copy only files which have been modified files since last backup, and write over the destination file without querying," as follows:

```
^<Backing up files A-D...^>
copy a*.* bak: /ae
copy b*.* bak: /ae
copy c*.* bak: /ae
copy d*.* bak: /ae
^<Please remove DISK 1 and insert DISK 2^>
^?
^<Backing up files E-J...^>
copy e*.* bak: /ae
copy f*.* bak: /ae
.
.
etc.
```

Any text between ^< and ^> is echoed directly to the screen. The ^? command stops and waits for the user to hit any key at which point the script continues.

You needn't worry about ZEX files being too large. Once they're loaded no further slowdown in your system is noticed. In any case, while they're being loaded you don't have time to get impatient as you're too busy watching the beautiful video screen display you've created!

This alphabetical ordering scheme ensures that every file is always backed up on top of its previously backed-up copy/version. This week's ALIAS.CMD is always copied over last week's ALIAS.CMD, and today's TCJ-COL.2 will always and automatically be copied onto the same disk as yesterday's copy.

I have over the years added to my BACKUP.ZEX the most beautiful graphics and box drawings of which my Qume 102 is capable, and combined with extensive use of its video attributes such as reverse video, blink, underline, and cursor off I have created what I feel to be the equal, in terms not only of functionality and power but of beautiful screen display, of any MS-DOS backup program. Of course there's one additional factor that in my eyes makes it even better: I wrote it myself!

I should mention as an aside that a very large number of popular and common CP/M terminals are capable of a great deal more graphics, boxes, windows and other fancy video displays than we normally use or even suspect. These features often remain unknown and unutilized simply because they were never standardized among the great variety of CP/M-style terminals. There is nothing, however—I repeat: **nothing**—to keep an individual user from creating for himself quite and surprisingly beautiful, if not transportable, screen displays. ZEX scripts, with their "^<text display^>" command, along with ZCPR3's ECHO command, give the user a great opportunity to do this, and many of my ZEX files and ARUNZ aliases are real beauties. It is especially unfortunate that these fancy features go unused in our CP/M-compatible world; the lack of "graphics" is always being touted as one of the reasons for people's decision to leave their old CP/M box behind.

Years ago Dennis Wright wrote a set of assembly language routines, GRWLIB and GRXLIB, that produce extremely sophisticated and beautiful graphics windows and menus, the windows capable of overlaying each other, appearing and disappearing, with text inside, etc. Similarly we already have one set of graphics windowing libraries for Turbo Modula-2, and Mr. Edward Jackson of California will soon be releasing another. I have used these Modula-2 routines to write COM files on my little ol' Morrow that rival displays I've seen on early Macintoshes and video arcades. I would urge users to take with a grain of salt claims for any alleged "lack of graphics" in CP/M and to incorporate these library routines into their programming. The beautiful displays really make a difference and greatly add to a finished product.

In any case, the alphabetical ordering scheme I use in my BACKUP.ZEX, along with the beautiful graphics I have coaxed out of my terminal with it, has not only proven eminently satisfying but has had the unintended (but very welcomed) side effects of giving me an opportunity to spend many happy hours acquainting myself with my terminal's video commands at the same time as it sharpens up my ZEX skills...

In any case, until about six months ago I had felt my BACKUP.ZEX to be complete as described above. The alphabetical ordering scheme worked perfectly, and the artistic video displays and friendly messages I added to it made backing up a pleasure, certainly an important consideration in a task normally considered to be the quintessential drudge work. After adding the ability, as described in last issue's column, to display to the screen the date and time of the last backup (about which I felt and still feel very proud!) I could think of neither bell nor whistle to add to it.

The only thing that occasionally bothered me is when my backup disk would fill up. This situation is most annoying. PPIP would of course report the error, but considering I'm inside a ZEX file there's not much it can do about it. If the disk would fill up, for instance, during the backing up of the "B" files, I would then have to suffer through PPIP's attempts to copy all the unarchived "C" and "D" files, with each of these attempts virtually guaranteed to fail. Although PPIP is smart enough so that if it senses the destination disk is full during one of the copies in a PPIP B*.* BACKUP: command it will abort that particular wildcard operation, there's no way it can anticipate the NEXT commands, being here of course 'PPIP C*.* BACKUP:' and 'PPIP D*.* BACKUP:.' It would then of course blithely attempt to execute these even though the task was impossible.

How was it to know this? I had to patiently watch sometimes 5, 6 and 7 PPIP "disk full" messages during copies and not being able to do anything about it. If I would CTL-C during the particular PPIP operation that sensed the disk full, it would report a "User abort," terminate that particular PPIP command—and simply move on to the next.

Well, what do you expect? It's just a computer!

I had only one way out of it at the time. Each time my backup disk would fill up, I would despairingly watch it happen, poor PPIP torturing itself with trying, and when the script paused for me to change disks I'd CTL-C out of it and abort the entire ZEX script. I'd then manually erase on the backup floppy whatever files I determined unnecessary and obsolete, and restart the script from the beginning.

Though it being the late 80's and I knowing this isn't what computers are supposed to do, I couldn't really think of any way out of the situation. I mean after all this is on an obsolete Morrow computer running an esoteric and little-used enhancement of an antiquated operating system. What, after all, could I expect?

Operating under the principle, however, in my life generally as well as in my computer programming specifically that it ain't what you got but what you do with it, the Great Spirit of Computers seems to have paid me one of its irregular visits. It one day hit me: if PPIP is smart enough to know when a destination disk full, can't I get the operating system to know? This is ZCPR3, after all.

I keep hammering home to all who will listen that one of the unique aspects of ZCPR3 is the way it provides methods for all its separate components to leave messages for each other. This is the raison d'etre of the ZCPR3 message buffer generally and the error flag particularly, the latter of which is here the perfect messenger for our job.

I decided to modify PPIP, otherwise a ZCPR3 tool in only the most superficial manner, to set the ZCPR3 error flag under certain conditions. I added code so that it now sets said flag to 7 (this figure arbitrarily chosen) if it senses a destination disk full. This way we provide a hook, through the "IF ER 7" command, for ZCPR3, and not just PPIP, to sense the disk full condition—and then provide a way out.

While I was fiddling around with PPIP's internals I decided to also set the error flag if the user enters a CTL-C during a copy operation. Occasionally I would see a backup copy take place that for a number of different reasons I didn't want to happen, and I had always wanted some way to allow ZEX, and not PPIP alone to abort if this happened. Now I had it.

Finally, PPIP will now set the error flag to '9' if the destination directory has no remaining entries. PPIP19 with these above modifications is available on all Z-Nodes.

All that was necessary in the ZEX script was to add the commands IF ERROR;GOTO ERR;FI; after each invocation of PPIP as follows:

```
copy a*.* bak: /ae
if er
    goto err
fi
copy b-.* bak: /ae
if er
   goto err
fi
 .
 .
 .
```

After I had modified PPIP to set the ZCPR3 error flag I then created a routine at the bottom of the script and began it with the label ';ERR' (ZEX labels begin with a semicolon). At this label there are one of two error messages, saying either that the backup disk was full (error flag set to 7) or that a user abort was entered (error flag = 8). (I didn't bother adding a "directory full" error message as that never happens on my floppies, though that could easily be done.) The user is thus given this message both by the program (PPIP) and the operating system (ZEX acting here as a proto-operating system.) The script would then properly abort as I wanted. No more helplessly watching unnecessary and impossible copies.

A listing of the error routine of BACKUP.ZEX is printed as Figure 1.

As it is fairly long I am not going to describe here the entire script. I want only to focus on the parts of it that relate to the sparks that went off in my mind consequent to the above modification.

The code I added above ('if err;goto err...) simply prints an error message and lets ZEX abort to the command processor. Having ZEX abort was the only way I could conceive of dealing with the situation. When the backup disk filled up I needed to do some work on my own, outside of ZEX, to erase unneeded files on the backup floppy as well as other things. I would never be able to anticipate which tasks I would need to perform. Aborting ZEX and returning to ZCPR3 provided an opportunity to do whatever I needed to. When I was ready to resume I would simply rerun the ZEX script.

But the Muse of Computer Automation wouldn't let go of me. Soon I started thinking: here I am using this incredibly powerful tool, suited to the automation of many tasks. Why do I need to exit from it? Can't I configure my ZEX script so that it will allow me to run a few commands **from the ZCPR3 command line** on my own, and then continue with the script? I had never before used ZEX precisely in this manner, to allow for user input from the ZCPR3 command line. If I could, however, it would let me do the above erasure of files or whatever is necessary so that the backup floppy will have enough room, and then return to the script.

This was an interesting challenge. ZEX has an "allow user input" directive (^"), which I naively at first thought might do the trick. I quickly realized, however, that this directive is primarily to allow user input **inside an application program**, not at the command line. No matter how I tried, no matter which internal ZEX command I experimented with I couldn't set it up so that ZEX would stop temporarily **while at the command line**, query me to enter a command, and then resume the script.

Where to turn?

The ZCPR3 utility CMD.COM popped into my head. After all, this is its function: to query for a command and then execute it. As you will recall from my first column, it is ideal for use inside a SHSET command line. However, I quickly discovered it is not meant for use within a ZEX script. ZEX will not stop at CMD and properly wait for a user command but rather will continue to send commands from the ZEX command stream. I am sure the reasons for this are very simple and logical, though they are at the present unknown to me.

Enter ARUNZ. In ZCPR3, when in doubt, look to ARUNZ. ARUNZ's "query for user input" command was a definite candidate for the job. I wondered if ZEX would allow this feature to operate. This is a greatly simplified explanation, but generally you insert the text you wish to display to the user between a $'' and a '', and then recall it later in the command line with $'11. A common use is:

```
$''Enter a command: ''
```

The user then enters his command or series of commands, complete with parameters if necessary, and then the entire string that the user entered is recalled wherever in the alias with the "$'11" command.

I tested an alias using this technique and it worked perfectly. An invocation of an ARUNZ "user input" alias inside a ZEX script allows you to pause the script, enter a command to be sent to the command processor, and then resume to the next commands in the script. Used inside BACKUP.ZEX this would allow me to enter whatever commands necessary to erase files from the backup disk, and then return.

Now that I had my tools my plan of action began to unfold as follows:

(1) Tell the user his backup disk is full.
(2) Store the name of the current directory in memory.
(3) Log on to the BACKUP: directory.
(4) Provide a directory listing to allow the user to see what files he has there.
(5) Provide a means to erase unneeded files, using whatever and however many commands the user sees fit: ERASE.COM, NSWEEP.COM, ZFILER.COM, etc.
(6) Return to the original directory.
(7) Resume the ZEX script where left off before the disk full error occurred.

In step five I realized that not only did I want to give myself freedom to choose whatever tools for erasing that I wished, but that I also wanted ZEX to load up one or two specified utilities that would facilitate that task for me.

Was this all possible? Item (7) seemed especially challenging. ZEX didn't keep any "pointers" to where it was in the script. This entire scenario certainly seemed to me much more elegant than having the ZEX script simply abort, but was it programmable?

First I began thinking about what commands I would like to run to facilitate the task of erasing files. Initially I thought it would be nice to run ZFILER which is generally the perfect tool for mass file deletions. In case, however, you haven't yet run into this situation (many of us discovered it with WS4), ZCPR3 shells cannot reliably be run from inside a multiple command line or ZEX script. In any case, I decided an approach utilizing (1) DIR/ERASE and (2) NSWEEP would be better. DIR and NSWEEP are better than ZFILER anyway in giving the user a better grasp of the relationship of the sizes of individual files to the total space remaining on the disk. ZFILER only allows display of individual file sizes with a manual Column for TCJ #38, May/June 1989"F" command.

I should point out here that Jay Sage's recent opinion that programs such as ZFILER and the like might perhaps be better

made ZCPR2 rather than ZCPR3 shells is right on point here; we would then be able to use it in our script if we wanted.

Temporarily putting aside the details of working out the challenge posed by step (7), I went to work. At the error routine label in BACKUP.ZEX I used the ^<text^> command (with fancy graphics, of course) to display a message indicating that my backup disk is full. The problem of storing (and later returning to) the current directory is taken care of by Paul Pomerleau's little-known PUSH11.COM, available on all Z-Nodes. Though ARUNZ has many symbols available for this purpose we are not inside ARUNZ and therefore need a standalone program. Entered with no parameter PUSH stores the current DIR:. Later in the ZEX script, when we are finished with our work on the BACKUP: directory, the command 'PUSH !' restores it.

I next log on to the BACKUP: disk and then do a DIR command, so I will have a listing of files in front of me.

Then going to my ALIAS.CMD I wrote an alias called INPUTERA which will serve as a preface for ERASE.COM:

```
INPUTERA $''Enter files to erase as a filelist: ''if ~nu <<
      $'11; erase $'11;sp;else;echo nothing entered.;fi
```

As discussed previously there is no reason we cannot run an ARUNZ alias from within a ZEX script. Though it occasionally presents far-ranging problems concerning the ZEX INPUT flag, that need not concern us here.

This will display:

```
Enter files to erase as a filelist:
```

on the screen and pause. If I now see from the DIR listing that files CHAPTER.1 and PROG.Z80 on the destination floppy are both 20k in length, obsolete and unneeded and obviously taking up precious space on the disk, I could enter:

```
CHAPTER.1,PROG.Z80
```

The ARUNZ script, which now has control, would then expand to:

```
ERASE CHAPTER.1,PROG.Z80
```

and my lovely but no longer wanted files would be unceremoniously deleted. (The question of how to ensure that one's backup floppies are continually updated—i.e. contain only files existing on the hard disk—is another story. I sometimes use Carson Wilson's CHECK28 program for this, which compares directories and shows which files are missing on one or the other, and sometimes a lovely ZFILER group macro script that Carson Wilson conceptualized.)

If nothing is entered at this prompt (''if nu $'11') then ERASE.COM is bypassed and the message 'NOTHING ENTERED' appears on the screen.

One proviso is in order here however, concerning the use of ARUNZ within a ZEX script. In writing BACKUP.ZEX I had the most interesting and unusual experience—from which, as usually happens, I learned immensely. At the point in the script where the aliases execute my computer locked up. Try as I might I could not figure out the reason for this. It was a particularly edifying experience, actually, because through it I learned that the normal technique I use to trace down bugs, the one that is considered the standard for programmers the world over, only works 99% of the time. That technique is of course the isolate- the-problem-by-stripping-away-extraneous-factors-one-by-one method. I have used this technique countless times and it has normally worked perfectly. After stripping away as many factors as possible, so the theory goes, you then add them back one by one and at some point—VOILA!—you find it. This was one time it didn't.

I isolated extraneous factors in this case by making an extract of the relevant part of the ZEX script and running that as a mini-script. This time the script worked, seeming to confirm the theory.

To make a long story short, however, I discovered, after many weeks, that the lockup was caused by a TPA problem. Unbeknownst to me—and perhaps to its author—type 3 ARUNZ running at 8000h will lock up when there is less than 39k TPA available to it on the system. Many users may think it completely impossible to ever drop down that low, but let me warn you: when you do work with ZEX scripts you may often be surprised at how low the mercury can drop on your TPA thermometer. ZEX.COM itself uses about 3.5k of TPA. On top of this you have to add the memory occupied by the script. BACKUP.ZEX is a bit over 8k in length. I start out with 50k TPA on my hard disk system; within this script I'm down to about 38.5.

The solution? Easy: it was precisely for these situations that the type 4 concept was invented. Make sure you're running ZCPR34 and use the type 4 ARUNZ, which has code to determine wherever is the most efficient place in memory for it to load and run. 8000h is obviously too high with only 38.5k TPA, so it simply relocates itself someplace lower in memory and runs perfectly.

Now we can load up NSWEEP. Control is returned to ZEX, where it sees an invocation of this honored utility. I should note in this regard that a friend and I have both found NSWP207 to mysteriously malfunction on drives E: and F:, so I use NSWP205 here. Since we have the 'ZEX input' flag turned to OFF we can enter whatever commands we want while inside NSWEEP, and ZEX is temporarily kept at bay. This allows us to see file sizes, get a feel for what's expendable and what's not, view files with an eye towards determining what can be erased, etc.

Now we come to the above-mentioned ARUNZ alias I wrote to give myself the ability to do that for which ZEX itself has no direct provision: to pause, allow the user to enter a command **from the ZCPR3 command line**, and then return to the ZEX script.

Our next ZEX command is then the ARUNZ alias INPUT:, which might be said to be a ZEX-compatible CMD.COM. It looks as follows:

```
INPUT  $''Enter any command ('EXIT' to exit  <<
       ZEX completely): ''$'11;inputret $1
```

INPUT simply displays its message on the screen, and whatever commands the user enters are stored into the ARUNZ symbol $'11 and then executed. I also want to make an allowance for a case in which something has definitely gone awry and the user absolutely needs to exit from the ZEX script. In that case the following ARUNZ alias EXIT does the job:

```
EXIT poke $+m0008 0
```

Eight bytes above the beginning of the message buffer is the 'ZEX control byte.' If that is poked to zero (as it normally is in non-ZEX operation) it shuts off the ZEX preprocessor completely and we are returned to ZCPR3. Incidentally, this effect is NOT realized by simply hitting the carriage return at the INPUT prompt, which would simply continue the ARUNZ alias and ZEX command sequence.

Although I could have put this 'EXIT' sequence directly into the original alias, I preferred to have it accessible from anywhere as an alias in its own right.

If 'exit' is not entered, then we are ready to give the user his last chance to enter any necessary commands, to definitely ensure that all unneeded files are erased from the backup disk and that there is sufficient room to continue the backup procedure before returning to ZEX. We therefore have INPUT chain to a second alias which I have called INPUTRET, which will indicate to the user that this is his last chance and then return to ZEX.

I should mention that a few days later, after I had added all the features I wanted to this ZEX script, INPUTRET looked a bit different, but at the time it was simply:

```
INPUTRET $'' Enter your final command ('or EXIT' to exit ZEX <<
        completely): ''$'11;if eq $'11 exit;echo returning <<
        to zcpr3...;push !;else;sak /p2;fi
```

—exactly like INPUT above except rather chaining to a second alias if the word 'EXIT' was not entered it would pause 2 seconds and return to ZEX. If 'EXIT' was entered, the EXIT alias is temporarily nested in, then 'PUSH !' restores the current directory and we return to ZCPR3.

We are now faced with the situation, as mentioned before, of wanting to return to the body of our ZEX script. So far so good; nothing special is required to do that. We certainly want to return to the part of ZEX outside the "error" routines from which we had just come, the part that does the actual copying. Probably, or so my original thinking went, we should use ZEX's ^: directive here, which reruns the command script from the beginning.

Wouldn't it be wonderful, though, if we could do the seemingly impossible: to return precisely to where we were before the error was issued (backup floppy full or whatever). But how could this be done? Unfortunately, for all its power one thing ZEX doesn't have is an "end subroutine" or "end procedure" command like actual languages do by which after a routine or procedure is jumped to and completed, control is then returned to where it was just past the jump (goto) command. Sure, we can use GOTO.COM, but ZEX has no internal "pointer" by which it can keep track of where it was just previous to the GOTO to allow it to return. Once a GOTO GOes TO someplace, there's no coming back.

Or is there?

If there's one thing I would hope ZCPR3 users get out of my work it is the power of the registers and other similar functions to pass messages back and forth to utilities.

I thought about it for a while—"registers" must have been going off in my head—and I knew how it could be done.

I went back to the body of the script and added two lines at each invocation of PPIP. I added before the beginning of each, a label whose name was the number that corresponded to the letter of the alphabet. Where I copy the "A" files became label "1", the "B" file copy sequence became label "2", and so on up to label "26" for "Z". Then I added an invocation of REG to set ZCPR3 register 1 (chosen arbitrarily) to this same value. The beginning of the script now looked as follows:

```
^<Backing up files A-D...^>
;=1
copy a*.* bak: /ae
if er
     reg s1 1
     goto err
fi
;=2
copy b*.* bak: /ae
if er
    reg s1 2
    goto err
fi
;=3
copy c*.* bak: /ae
if er
    reg s1 3
    goto err
fi
;=4
copy d*.* bak: /ae
if er
    reg s1 4
    goto err
fi
^<Please remove DISK 1 and insert DISK 2^>
^?
^<Backing up files E-J...^>
;=5
copy e*.* bak: /ae
if er
     reg s1 5
     goto err
fi
 .
 .
 .
etc....
```

Get it? Who cares if ZEX has no **internal** pointer? Our operating system is so superb it provides enough services externally! We use register 1 to be our **external** pointer for ZEX.

What we've done is use the REG command to set register 1 to a different value depending on where in the alphabet we are in our copying when the disk full message was given. If PPIP detects a disk full error during copy of the files beginning with the letter "A", PPIP itself sets the ZCPR3 error flag and then we set register 1 is set to "1". If the disk full is detected during the copy of the "B" files, register 1 is set to "2", and so on up to "26" for the "Z" files. These values are arbitrary; assigning the numbers 1 to 26 to the letters A to Z simply makes the most sense. In other words, if an error is detected, register 1 will be set to a value from 1 to 26 depending on which files we have just been copying.

"A ZCPR3 programmer isn't worth his salt if he ain't creative; the operating system sure gives us enough opportunities."

You may wonder how this information could possibly be useful to us. How can setting a register—a feature of the command processor—could be useful to ZEX which (unlike ARUNZ) has no internal symbol to represent the contents of the registers. Jay Sage has been meaning (when he pares his workday down to 18 from its current 20 hours) to add the entire ARUNZ symbol structure, which includes a symbol for the values held in the registers, to ZEX. Until that sweet time comes, however, we must be inventive. And a ZCPR3 programmer isn't worth his salt if he ain't creative; the operating system sure gives us enough opportunities.

Figure 1. LISTING OF BACKUP.ZEX ERROR ROUTINE:

```
goto end            ; if no error
;=err
zif
if er 7             ; CTL-C entered
goto errseven
fi
^[ZEX has detected that the BACKUP disk is full.^]
sak /p2             ; leave message on screen for 2 seconds
push
backup:             ; log to backup directory
dir
/inputera           ; run 'ERASE' alias
^[^|^|Loading NSWEEP might help, so you can view your files...^|^]
ns                  ; NSWEEP
/input
/inputret
push !
rs goto $$r1        ; return to where we left off
;=errseven
^[^|Abort request detected, ZEX script terminating...^|^]
zif
;=end
```

Enter (1) RESOLVE, and (2) one of my favorite ruses, ye olde "numbers-as-a-character-string" routine, delved into in some detail in my last column.

Pay attention to the ZEX labels I've added, which have the form ; = labelname, that I have inserted before each COPY command. The name of the label before copying files beginning with "A" is "1"; the label just preceding the copy of "B" files is named "2", and so on through Z. A ZEX label name, just as any filename or in fact most any other name we use in computers, may consist of any alphanumeric characters and there is no reason we cannot use digits instead of letters. It's the same scheme we use with setting the register. At the label named "3", we copy files beginning with the letter "C" and upon an error at this point set register 1 to 3. We will see shortly how handy this is.

After the INPUTRET alias has taken us back to our ZEX script, the next command in BACKUP.ZEX is:

```
RESOLVE GOTO $$R1
```

Remember that since we are in ZEX a double dollar sign is how we represent a **single** dollar sign; ZEX strips out the first. So we are left with '$r1'. Of the many things that RESOLVE does one of them is return the value of any of the ZCPR3 registers—and for this we should be grateful or else this entire procedure could not work. '$r1' therefore expands to "the value held in register 1."

Now let's go back to our script and simulate a real-life situation. Suppose we are backing up our MEX: directory and we are on backup disk number 4. While PPIP is copying the files beginning with the letter "S" it runs into a full destination disk. It sets the ZCPR3 error flag and then our ZEX script sets register 1 to the value of 19. ZEX then jumps to the error routine at the bottom of the ZEX script. We are told the disk is full, we get a directory and are asked which files we'd like to erase. We erase a file or two here. NSWEEP loads and since we are able to use its V)iew command we see a few more files that are redundant, though we're not absolutely sure yet we have enough space on the backup disk to confidently return to the ZEX copying routines.

We next take advantage of the INPUT alias, which allows us to run any program of our choosing, to run Carson Wilson's FD which gives us a better sense of which files on the disk are most recent. We use this new information about datestamps to determine that there is one very large and old file for which we no longer have any use, and therefore at the final, INPUTRET prompt erase it. We are then finally returned to ZEX. At this point the next command ZEX encounters is 'RESOLVE GOTO $$R1'.

Since register 1 is set to 19, this command will expand (or "resolve") to "GOTO 19", which means go to the ZEX label named "19". Again, there is no reason we cannot have a label named "19", even though label names are **usually** composed of letters and not numbers. Our label "name" not so coincidentally happens to be the numerical value held in ZCPR3 register 1, and this suits RESOLVE just fine. And—what do you know—label 19 just happens to be at the "COPY S*.* BAK: /AE" command. So GOTO returns us exactly to where we were in the ZEX script before the error occurred.

Neat, huh? I had done what I'd set out to do.

Never one to give myself any rest, however, a day or so later the thought came to me (a classic symptom of that dreaded disease that programmers get, "feature-itis"): wouldn't it be nice if we could get a message flashing on the screen:

```
RETURNING TO THE ''S'' FILES...
```

just before the actual return? Sure, I said; why not?

But how to do it? How, where, and what could possibly send to us this "S," and how would we store, keep track of and later access it? Could PPIP possibly send this letter to us somehow? If not PPIP, then what? I challenge anyone who doesn't think in terms of RESOLVE and shell variables to come up with a solution, short of writing a new program...

The idea is to create a *.VAR file that matches letters with numbers. In other words, we're going to set up a situation in which when RESOLVE finds the string '%19' it will give us the letter 'S'. Here's what we do. We're going to define some variables into our SH.VAR file. This time I'm not going to bother defining another *.VAR file as the current one.

We use SHDEFINE as "SHDEFINE SH." Note that with SHDEFINE even if another variable file is currently defined to the system we can specify as a command line parameter the particular file to which we would like to add or replace variables. Using SHDEFINE's E)dit command we add the following variables and definitions, starting from where variable '1' = definition 'A' and going all the way to variable '26' = definition 'Z'.

VARIABLE NAME	DEFINITION
1	''A''
2	''B''
3	''C''
4	''D''
5	''E''
6	''F''
.	
.	
.	

and so on, until 26 and "Z". Now, we can add one more command to our INPUTRET alias described above (RESOLVE.COM is always renamed to RS.COM):

```
RS ECHO RETURNING TO WHERE WE WERE-AT THE %$RF01 FILES...
```

and the entire alias becomes:

```
INPUTRET $'' Enter your final command ('or EXIT' to exit ZEX <<
      completely): ''$'11;if eq $'11 exit;echo returning to <<
      zcpr3...;push !;else;rs echo returning to where we  <<
      were-at the %$rf01 files...;hold 2;fi
```

Don't worry about the length of this alias, even if with your editor it exceeds 200 characters, the number normally associated with the limit of the command line buffer. Any text in the 'user input' part of the alias is **not** placed in the command line buffer and should therefore not be counted towards the 200 characters.

Can you understand this alias? It is similar to, but slightly different from the RESOLVE GOTO $$R1 command we put in our ZEX script. Here we are relying on ARUNZ's and not RESOLVE's register-expansion facilities, and the syntax is slightly different. Here in an ARUNZ script in which we want to represent "the contents of register 1" we must follow the "R" (stands for Register, of course) with an "option letter," followed in turn by the register number. Our "F" option means "floating decimal," whereas if the number is one digit long, we want it represented as one digit only, and if two digits it should appear as exactly two, and similarly with three. In other words, don't give us any leading zeroes, only the raw number. When RESOLVE returns the contents of a register it never puts leading zeroes; ARUNZ's option letters allow it or not.

By the way, it should be noted that RESOLVE cannot expand both the register value AND the ensuing and contingent string variable (that the register value then becomes). Therefore, we cannot put this command inside the ZEX file; we must put it into ARUNZ where we can rely on ARUNZ's register expansion symbol and leave the variable expansion up to RESOLVE. Unlike in our first use of RESOLVE where it expands the register contents, here ARUNZ does so.

With our register 1 still set to 19 ARUNZ resolves the register symbol '$rf01' and our command then becomes:

```
RESOLVE ECHO RETURNING TO WHERE WE WERE-AT THE %19 FILES...
```

Now RESOLVE's variable-expansion, and not its register- expansion facility comes into play. ARUNZ has sent it the string "%19." Remember that when RESOLVE sees a '%' it knows that a named shell variable is to follow. It knows therefore to look in the currently defined shell variable file, defaulted to SH.VAR unless we have defined it otherwise with SHFILE.COM (we haven't), and determine the definition that has been given to that variable. That definition is then returned to us for our use. So RESOLVE then looks into SH.VAR for a variable named "19". And what does it find? Of course—the letter 'S'.

Our command line is therefore expanded to

```
ECHO RETURNING TO WHERE WE WERE-AT THE ''S'' FILES
```

ECHO presents this lovely and informative message to us on the screen, we are tickled pink that our operating system is so elegant, and finally control is passed back to the exact point in ZEX where we were when the disk full message first occurred and where RESOLVE, GOTO, ZCPR3 registers and ZEX labels team up to make us feel incredibly grateful for this work of art called ZCPR3.

Before closing, I'd like to mention that in the last few months I have been experimenting further in the vein of last column's topic, writing more and more programs to store various dates in shell variable files. As a result I have come up with something I believe will be very handy for users of the PC-Pursuit network. Until now monthly use on PCP has been unlimited. Beginning on May 1 however that will all change. There will then be a monthly limit of 30 hours, after which any use will be charged on a per-time basis.

I am sure that users of MS-DOS communication programs have been busily rewriting their scripts and utilities to be able to keep track of their monthly time so the user will be alerted when approaching the 30 hour limit. Now for those ZCPR3 users whose systems include datestamping, a real-time clock and the extraordinary modem program MexPlus about which I wrote a bit last time, I have updated my PCP MexPlus script so that it will keep track of monthly PC-Pursuit time. It stores hours and minutes temporarily in the ZCPR3 registers; when the computer is turned off they go into (where else?) shell variable *.VAR files. Thanks to ARUNZ' date symbols, the *.VAR files are changed each month and the time count begins anew.

Those who are interested in this script, or for that matter anything else about which I write in my columns, may write me care of TCJ. I'd also be glad to send a copy of the entire BACKUP.ZEX script, either the generic version or my personal copy configured especially to take advantage of the extended graphic and video capabilities of the Qume 102a terminal.

Z you next time... ■

Real Computing

(Continued from page 32)

For code references, the process is a little more complicated. You see, each module has its own code requiring its own static base to be pointed to by SB. Therefore, each entry in the link table specifies the module entry and the offset from that module's code base (so each module can have multiple entry points). This has to be filled in by the loader. The calling process has to use the Call eXternal Procedure (CXP) instruction. This instruction saves the current PC and MOD (module register) on the stack. It then loads the MOD register from the link table entry, loads the SB register from the first doubleword of the module table entry, and loads the PC with the module entry's code base (third entry) plus the offset. For example:

```
.EXTERN   PRINTF
JSR       PRINTF
```

The assembler recognizes the call to an external procedure and allocates an item in the link table. The actual code generated is:

```
CXP       5
```

The assembler can simply fill in zero in the link table entry. When the module is loaded, the system fills in the module table entry address of the module containing PRINTF (low order 16 bits), combined with the offset of PRINTF within that module (high order 16 bits), in the link table entry.

The module which is called must be aware that there are four extra bytes on the stack. It also must return with an RXP (Return from eXternal Procedure) instruction instead of a normal RET. Because any global module might be called by some other module, any callers within the same module must use the external mechanism to call the procedure, even though the procedure is in the same module. There is a small performance penalty associated with the use of CXP/RXP, so any subroutines not requiring external visibility should be coded local ("static" in C jargon) and called with BSR.

Evidently, the loader must be endowed with some intelligence. It has to keep symbol tables listing the global symbols of the various modules in each task image, including that of the operating system. In the MMU environment, tasks cannot call modules in other task images, but if the symbol tables were kept along with the modules, the OS could conceivably allow modules, such as run-time libraries, to be shared. Care would have to be taken to ensure that such modules were reentrant, either by avoiding static variables or by using semaphores or other such techniques.

Interesting, isn't it? Describing a feature of the NS32 hardware, I moved seamlessly into how the operating system should make use of it. This just shows one of the ways that the NS32 architecture has been intentionally designed to make software easier to write. How different from the Intel and Motorola architectures, where the programmer must wrestle with the processor to get around its "features"!

One problem shared by all architectures in generating relocatable code is the use of address constants within programs, for example, within initialized data. The usual (and ugly) mechanism for handling this problem is to use a bit map for the module, with a bit set for each location which needs to be relocated in some way.

Next time

Next time we'll examine the 32532, the highest-performance 32-bit microprocessor available. We'll also look at the 32CG16, a version of the 32016 with high-performance graphics commands built right into the instruction set. And we'll check in on our hardy band of operating system writers. Until then, may your core not dump and your process not zombie. ■

Real Computing

The National Semiconductor NS32032

by Richard Rodman

The PC of the 90s

The personal computer today has reached a technical plateau, much like the plateau achieved in 1980 with the Z-80, 64K RAM, and CP/M. Like back then, though evolutionary schemes exist, these cannot succeed. Why not? The defining standard is 8088 and 640K RAM, and that standard cannot be discarded without complete industry-wide agreement. Such agreement has never happened and will not happen. Instead, a single leader will design a machine that is so much better that people will abandon the old machine for the new.

The defining characteristic of such a machine is that it uses virtual memory, so that the machine can run any programs for it regardless of how much memory is installed. Likewise, it should inherit the characteristics of previous machines, that all software should run regardless of what kind of graphics, keyboard, disk drives et al. are installed.

Secondarily, we would expect that the machine should be multitasking. At a minimum, the user should be able to "set aside" what he is doing to do something else, then pick up where he left off.

One would expect a graphical user interface that would allow the "set aside" operation to be implemented as a "collapse to icon"-type of action. It would not, however, be Macintosh-like.

A mostly unrecognized problem with today's PCs is their boxy, clutzy, cheap and inelegant cabinetry. The next generation machine has got to fit in better, both in office decor and in the home. It ought to look like a stereo.

Further, it ought to interface well with what the user already has. It ought to have audio in and out with phono jacks, connections to telephone lines, and video in and out jacks. Personal computers have been moving in the direction of more and more exotic video monitors and interfaces, which is the **wrong** direction, especially for home computers. The new machine should feature high-quality composite video with anti-aliasing, good gray scales, full interlace, usable with VCRs, and built-in genlock. They ought to tie in with the user's video and audio equipment.

The RS-232 connections should follow today's trend of using modular connectors. Unfortunately, there is no standard for these connections. I propose using 6-pin connectors with symmetric connections so that data cables would always be "crossed" and anything could be connected to anything:

```
Pin 1-Handshake out (DTR)
    2-Data In
    3-Ground
    4-Ground
    5-Data Out
    6-Handshake in (DCD)
```

What do people love to control their VCRs, TVs and CD players with? A mouse? A trackball? No, they use a handheld infrared keypad. Similarly, while the graphic interface could allow the use of a mouse or trackball, the basic control should be through a handheld infrared keypad. If you're seated at the computer, you can use the keyboard or a small wire-connected keypad.

The use of a keypad does not preordain simple numeric menus. No, even on a Macintosh screen, there are only so many items which could be selected, so many actions which can be performed. The interface can be done much more simply and cleanly than that of a Macintosh.

I suppose the real danger in making computers that easy to use is that the machine will be a commodity item, and we'll be flooded with machines from Korea and Singapore, but the real challenge in this machine will be in making the software so clean and fluid that its operation will be "noiseless" and "colorless". The simplicity of operation will conceal considerable complexity.

The NS32 Module Table and External Addressing Mode

One of the most interesting features about the NS32 processor is the built-in support for dynamic binding, that is, linking of multiple modules at run-time. It does this by means of a module table.

The module table must be located in the first 64 kbytes of memory. Each module in the system has a 16-byte entry in the table. This entry is called the "module table entry"; the module table entry of the currently executing module is pointed to by the MOD register. Since the MMU can page this memory, the actual table can be much larger, but there is an absolute ceiling of 16,384 module entries per task image.

When a module is loaded, a module table entry is created, and the entry's first three doublewords are filled in with (1) the address of the module's static base, or data segment; (2) the address of the module's link table; and (3) the address of the module's code base, or code segment. The object code itself should always be coded to be address-independent, so that static data references will be relative to the SB (static base) register, and code references relative to the PC (program counter).

But what about references to symbols in other modules? This is what the link table is for. Each link table entry is a single doubleword.

For data references, the assembler generates a reference to a link table entry. When the module is loaded, the loader fills in the actual address in the link table entry. The object code instruction then performs an indirect reference through the selected link table entry. For example, let's say there is a buffer called BUFFER somewhere in the system. The programmer codes:

```
.EXTERN   BUFFER
ADDR      BUFFER,R0
```

The assembler recognizes this and allocates an item in the link table. The actual code generated is:

```
ADDR      0(EXT(4)),R0
```

The assembler can simply fill in zero in the link table entry. When the module is loaded, the system fills in the address of BUFFER in the link table entry.

(Continued on page 31)

ZSDOS

Anatomy of an Operating System

by Harold F. Bower and Cameron W. Cotrill

Harold F. Bower, Major, US Army Signal Corps; BSEE, MSCIS, Ham (WA5JAY), avid homebuilder (starting with 8008 running SCELBAL).

Cameron W. Cotrill, Vice President, Advanced Multiware Systems; specialist in "impossible" real-time hardware and software systems.

In the first part of this article, we presented the philosophy and the features of ZSDOS (Z-System Disk Operating System). In this portion, we will summarize the performance of ZSDOS, share a few of the tricks we used to shoehorn all these features into 7 bytes, and give a few programming examples showing how to use some of the new features of ZSDOS and ZDDOS.

ZSDOS Performance.

Measuring the performance improvements of ZSDOS is a complicated matter. During development, an entire suite of tests was run on ZS/ZDDOS in various configurations in an attempt to validate the design tradeoffs. The most revealing tests of BDOS differences turned out to be a series of assemblies done under control of a command script. This should be no surprise as assemblies are by nature disk intensive.

To reduce the perception that our results are "tailored" or skewed in favor of a particular system or configuration, different processor chips (Z80 and HD64180), different BIOSes (MicroMint, XBIOS, Ampro), and different media (RAM disk, Hard Disk and Floppy disk) were used in the timed runs. Since the results were most affected by the media, results are shown in the categories of RAM, Hard Disk and Floppy Disk performance. No form of file date stamping was done since ZSDOS would have a distinct advantage in this field.

Three sets of hardware were used in these analyses in an attempt to minimize the effect of any unique processes in a given system from skewing the results. The first system (System 1 in the timing runs) was a "stock" MicroMint SB-180 operating at a 6.144 MHz clock speed. System 2 was an Ampro Little Board 1A with a Z80 running at 4.0 MHz, and System 3 was a homebrew Z-180 system designed to be compatible with the SB-180 operating at 9.216 MHz. Complete information on each system is given in the Appendix.

Operating Systems.

CP/M 2.2. Gary Kildall and Digital Research developed this operating system for 8-bit processors in an evolutionary process on early 8080-based computers. A subsequent product, CP/M Plus (also known as CP/M 3) is still in limited use, but has not gained the wide acceptance of the earlier release. CP/M 2.2 is coded in 8080 assembly language and is a non-banked, non-reentrant single-user, single tasking operating system.

ZRDOS 1.9. Echelon Incorporated released many versions of this CP/M 2.2-compatible operating system over the past several years. It is coded in Z80 assembly language and will therefore not execute on 8080 processors. Some additional features were added, such as one-level reentrancy under user control, and return of the current DMA address. Later versions (after 1.5) include enhanced support for hard disk media by not rebuilding the allocation bit map on a disk relog command. Version 1.9 added larger disk and file sizes. Like CP/M, it is single-user and single-tasking.

ZSDOS. This is the topic of this article, with details and descriptions of features contained in Part I. ZSDOS is coded in Z80 assembly language and is also a single-user, single-tasking operating system capable of single-level reentrancy.

Since this report was an aimed at formalizing an evaluation of the performance characteristics of ZSDOS, a number of different variants to the above operating systems were initially timed. Because the performance of these systems was very similar to others in the test, their comparative results are simply summarized below.

CP/M 2.2 with Plu*Perfect Systems' PUBlic patch. Only minor differences in performance from the basic CP/M 2.2 were noted, so results of the patched system were not included in the final results.

ZRDOS 1.2. The performance of ZRDOS 1.2 was very close to CP/M 2.2, being a couple of percent slower in the majority of cases. It was therefore not included in the final timing analyses.

ZRDOS 1.7. Timing tests indicate no significant performance differences between ZRDOS 1.7 and 1.9.

ZDDOS. Since ZSDOS and ZDDOS are largely the same code and since comparative timings between them show less than a 1% difference, only times for ZSDOS will be presented.

BASIC I/ Systems (BIOSes).

MICRO MINT, SB-180. While MicroMint currently ships Version 3.2 with their systems, a slightly modified version of 2.7 was used in these timings on the SB-180. The changes included independent step rates for floppy drives, different floppy formats and fixing of eight-inch drivers as well as a slight amount of optimization. Little performance difference from the standard BIOS should be noticed. A 54k system size was used. The BIOS uses programmed I/O on most peripherals with DMA functions of the 64180 processor used for Floppy and RAM disk data movement.

XBIOS, SB-180. XSystems' XBIOS version 1.1 is an extremely powerful and flexible banked system with excellent tools and interfaces. Malcom Kemp has concentrated on providing functions in this release, and has deferred optimization to future releases. XBIOS fully supports the ETS180 IO+ board, allows complete configuration of peripherals, and provides a larger TPA since only a small kernel resides in the primary memory area. Most of the BIOS code resides in an alternate memory bank. XBIOS installs the largest possible TPA when used which was 57.5k for these tests. XBIOS was installed with three buffers for disk I/O.

AMPRO, Little Board-1A. A stock version of the Ampro version 3.8 BIOS assembled with no ZCPR support was used for testing. A system size of 59k was chosen to provide support for 5 hard disk partitions spread over two physical drives. NZCOM was then loaded to provide Z-System support. The Ampro BIOS is strictly a polled system and uses no interrupts or DMA.

Evaluation Procedures.

Since the goal of evaluating performance was to heavily exercise BDOS functions, a set of fourteen assembly modules, thirteen of which were 2 to 4k in size, and one of 6k were assembled to produce Microsoft REL files. To restrict external influences, no file date stamping was used, and many ZSDOS features such as Public and Path were disabled. On the other hand, to provide a semi-realistic setting, ZEX.COM and the executable assemblers were placed in a different Drive/User with the ZCPR search path set to locate the files on the second directory scan. SLR's SLR180 assembler was used on system 2, while tests on systems 1 and 3 used Z80ASM+. Assembly was done under the control of a memory-based SUBMIT utility (ZEX Version 3.1A) script file. Times were measured from the carriage return terminating the command invoking the ZEX file to display of the "Done" message after assembly of the last file. After each run, the .REL files produced by the assembly were erased so that the same disk space could be used in the next run. No other files were added or deleted to any media during the timing runs. At least three runs were performed for each configuration, and the results averaged. Timing was manually performed with a stopwatch.

Due to the radical differences in access times for different media, three categories of times were considered; RAM disk, Hard Disk, and Floppy disk. If you think you know how each system fared, read on—there may be a twist or two in the plot.

RAM DISK

The Ampro has no RAM disk, so timings in this category reflect only the SB180. The SB180 computer is equipped with 256k of memory. The standard MicroMint BIOS divides this into a 64k main memory area and a 192k RAM disk. With XBIOS as tested here, 64k is allocated for the main memory, 24k for the banked portion of XBIOS, buffers and banked system extensions. The remaining space is available for a RAM disk. RAM disks on the SB180 use built-in DMA capabilities of the HD64180 processor to move "sectors" of data rather than the slower block move instructions used by Z80 systems.

Exiting a program via the Warm Boot vector in CP/M relogs the A drive. To minimize time penalties imposed by this, a Hard disk partition was defined as the A drive. Needed programs as well as the assembly modules were placed on the RAM disk (M:), with ZEX.COM and Z80ASM+.COM placed in User 15 and the sources files in User 0. The search path for this phase was: Drive M, User 0 to Drive M, User 15.

Since the RAM disk is defined as a non-removable media in the Disk Parameter Block, the "Rapid Relog" feature of ZSDOS and ZRDOS was expected to produce much shorter execution times than CP/M for this series of measurements. As can be seen from the results, this was indeed the case. The raw timings in seconds with percentage changes from the shortest time are:

	ZSDOS	ZRDOS 1.9	CP/M 2.2
BIOS 2.7	17.0 (---)	17.1 (+4%)	36.4 (+114%)
XBIOS 1.1	14.2 (---)	14.5 (+2%)	34.5 (+144%)

The effects of the Rapid Relog feature were borne out, with ZSDOS being a couple of percent faster. Disabling the Rapid Relog feature of ZSDOS produced nearly identical results to CP/M, so most of the additional time for that system may be attributed to rebuilding the disk allocation bit maps for Drives A and M on each warm boot.

Hard Disk

Three systems, 6.144 MHz SB-180 (System 1), 4.0 MHz Ampro Little Board-1A (System 2), 9.216 MHz Z-180 Homebrew SB-180 (System 3), were used to gather information for this phase. This latter system was added to demonstrate performance on a heavily loaded system.

	ZSDOS	ZRDOS 1.9	CP/M 2.2
1-BIOS 2.7	0:54.7 (---)	1:16.6 (+40%)	1:34.7 (+73%)
1-XBIOS 1.1	0:52.2 (---)	1:15.4 (+44%)	1:33.4 (+79%)
2-AMPRO	1:55 (---)	2:44 (+43%)	3:15 (+70%)
3-BIOS 2.7	1:07.7 (---)	1:40.6 (+49%)	1:50.2 (+63%)
3-XBIOS 1.1	1:29.5 (---)	2:06.4 (+41%)	2:11.3 (+47%)

As in the previous RAM Disk results, the results of ZSDOS with "Rapid Relog" disabled and CP/M were nearly the same confirming that rebuilding the allocation bit maps on a disk relog is the principle cause for the increased CP/M times.

All reported times were made with a path which forced a search of the current directory before locating executable files on the second path element. As an experiment, the path on the Ampro system was changed to go directly to A2:, eliminating the current directory scan. All DOSes showed an identical 10 second speedup, indicating directory scan time for all DOSes was the same.

A further point to note is the effect of multiple disk buffers on performance. For system 1, the number of buffers was adequate to retain directory information which improved performance over the single-buffer Micromint BIOS by 1 to 5%. In system 3, the buffering was inadequate to retain necessary information, so the multiple buffers were of no benefit.

Floppy Disk

Examination of system performance on a Floppy Disk system was tailored to duplicate, as closely as possible, a hypothetical operating configuration using multiple drives with non-trivial search path along differing Drives and User area lines.

Since all three primary operating systems of interest to this analysis (ZSDOS, CP/M 2.2 and ZRDOS 1.9) rebuild removable-media disk allocation maps on a relog, there was no need to explicitly disable the "Rapid Relog" feature of ZSDOS for this portion of the study. Results are:

	ZSDOS	ZRDOS 1.9	CP/M 2.2
BIOS 2.3	2:18.7 (+2%)	2:22.4 (+5%)	2:16.0 (---)
XBIOS 1.0	2:29.5 (+0.5%)	2:32.7 (+3%)	2:29.0 (---)
AMPRO	2:26 (+1%)	2:28 (+2%)	2:25 (---)

Since all of the operating systems are functionally identical in a Floppy Disk configuration, we did not expect large differences in measured times. We were therefore not surprised with variations over a spread of only five percent. While we strove to make ZSDOS as efficient as possible, CP/M was still the champ on floppy systems by a nose.

As a final comparison test between the three DOSes, the amount of time WordStar 4 took to ^QC and ^QR through the 92k ZSDOS source file was measured under all three DOSes. All timings were within 1%, indicating that read/write to open file times were similar.

Performance Conclusions

ZSDOS offers significant improvements in system performance on CP/M 2.2 compatible Z80-compatible computer systems with fixed media even under the restricted test conditions which disabled some of the most powerful features of ZSDOS. Even more impressive results may be obtained in a "tuned" installation with such features as Public files, and proper selection of the DOS search path (improvements of 9% on a hard disk system are typical).

The other major conclusion that can be drawn from this effort is that the selection of a BIOS tailored to the requirements is

crucial to achieving optimum performance. The multiple buffering capability of XBIOS offers speed increases in systems where an adequate number of buffers exists, but degrades floppy-based and heavily loaded hard disk performance.

During the data gathering for this report, an anomaly was noted with respect to CP/M Plus (or P2DOS) stamps. System #1 was initialized for P2DOS stamps on the disk holding data files to quantify the differences. In all cases ZSDOS was affected less than one percent, yet ZRDOS increased to seven percent longer than ZSDOS on RAM disk, 20% longer on floppy and 144% longer on hard disk. CP/M 2.2 was similarly affected, but to a lesser degree, increasing times over ZSDOS to 115% on RAM disk, ten percent on floppy and 140% on hard disk. While neither ZRDOS nor CP/M 2.2 can manipulate this type of stamp, merely using a disk which is so prepared will result in slower processing.

How We Did It

During the year or so that we pursued our independent paths in modifying H.A.J. Ten Brugge's excellent P2DOS alternative to CP/M 2.2's BDOS, our approaches were somewhat diverse. While Cam's approach was directed at perfecting features, Hal's effort was directed at streamlining the code to create a "speed demon" operating system, and Carson concentrated on enhancing embedded Date Stamping. In mid-1987, Bridger Mitchell was instrumental in getting us to pool our resources and collaborate in a joint venture. The results have been more than worth it. In Part I, we described the functional enhancements and standards embodied in ZSDOS, and have just shown the performance improvements compared to CP/M 2.2 and ZRDOS 1.9. In our efforts to foster better code for our 8-bit systems, we would now like to describe how the task of adding features and decreasing execution time was accomplished without increasing the Operating System memory requirements.

The topic of code optimization is a controversial one. In the early days of computers, programmers were saddled with small memory space and slow processors, so every effort was made to optimize programs for speed and size. As memory became cheaper and processors emerged with ever increasing clock speeds, programming techniques became lost to all but a few. This same path of evolution has also been followed in the Personal Computer field.

To demonstrate this point, first compare the 3.5 kbyte CP/M 2.2 BDOS and the 1 kbyte Plu*Perfect DateStamper to the functionally superior 3.5k ZDDOS. Next, compare the 3.5 kbyte size of CP/M 2.2 and ZSDOS to the 16 kbyte size of the functionally similar MS-DOS 2.1. To carry the point further, contrast the almost 16 kbyte COMMAND.COM to the 7 kbyte size of a more capable ZCPR3 Command Processor with a full environment. Some of this bloat is understandable with the change in processor chips. On the other hand, the more powerful instructions of 16-bit 808x processors should have counteracted a good portion of this code bloat.

In line with the size comparisons, execution speeds also suffer with the larger code. Friends and co-workers who are used to working with PCs and clones operating at 4.77 and 8 MHz clock rates are constantly amazed at the speed of even a lowly 4 MHz ZSDOS system, and dazzled at the 6 and 9 MHz Hitachi 64180 systems running the same software! While much of this is subjective, quite a bit is due to the fact that the "smaller" 8-bit code has been hand-coded and optimized, whereas the PC arena is devoting more of its energy to coding in high-level languages. This makes sense under certain circumstances (e.g. during development and for long-term maintainability), but it most certainly does NOT make sense for operating systems where size and speed are of the essence.

Since all of our efforts have been directed at the Zilog Z80 and compatible family of microprocessors (including Hitachi's 64180 and National's NSC800), the optimization steps covered here apply directly only to these. Having stated that, we also need to point out that many of the basic concepts will still apply to other processors, although details may differ.

No matter what processor is used, the goals of faster program execution and smaller memory size are in conflict. Smaller memory size normally means using each section of code as many times as possible—typically by using many subroutines. Faster code execution often means avoiding as many subroutine calls as possible. In every program undergoing optimization, the conflicting size and speed requirements must be balanced. This balance can be highly subjective. In ZSDOS, code size was the primary concern though significant effort was given to making the smaller code run as fast as possible.

Now for the minutiae. If you are not a programmer, or are interested only in how to use ZSDOS, you might want to skip to PROGRAMMING FOR ZSDOS. For the diehards—here it is!

One of the first techniques we used in optimizing code was to examine all JUMP instructions. The basic instruction is three bytes long and executes in 10 clock cycles on a Z80. These absolute jumps may be unconditional (JP addr), or conditional (JP C,addr) based on the contents of the Carry, Zero or Parity/Overflow flags. The Z80 also features a two-byte Relative jump (JR) which also may be absolute (JR addr), or conditional (JR C,addr) based on the Carry or Zero flags. The relative jump is only two bytes long and may branch only to addresses within the range of +127 to −128 bytes of the jump instruction. While it is relatively easy to blindly change all jump instructions within range to Relative jumps, the careful programmer will also note that the Relative jump may carry a time penalty. The absolute relative jump, and conditional jumps where the condition is satisfied (the jump is taken) require 12 clock cycles compared to the long jump consuming only 10 cycles regardless of condition. On the other hand, conditional relative jumps need only 7 cycles if the condition is false. This type of optimization was one of the first used in our efforts to enhance P2DOS.

The next simple optimizing technique we used was to make maximum use of the Decrement-B and Jump Relative if Not Zero (DJNZ) instruction. This two-byte sequence executes in 8 or 13 clock cycles (B=0 and B<>0 respectively) for an absolute time and code saving over separate decrement/jump sequences. In some of our work on ZSDOS, using this instruction required redefining register usage to free up the B register for use as a counter.

Another simple optimizing step was examining the use of the IX register. IX holds the argument passed to DOS in the DE register (typically a file control block pointer). Despite having this value available all the time, there were a significant number of cases when faster and/or shorter code was produced by moving the pointer into HL. This was normally the case when the same offset within the FCB was accessed two or more times in succession.

The final "simple" optimization technique we used was to examine all PUSHes and POPs to the stack and delete any found to be unnecessary. While this sounds simple, it is quite a chore in a complex program such as ZSDOS where CALLs call other CALLs which call still other CALLs, etc. Each path must be examined to insure that the registers are, in fact, not altered or needed.

After the above "simple" optimizations were performed, A series of what we term "moderate" optimization steps were undertaken. One of these involved examining all series of sequential checks on a byte (such as the input command character scanner) and structure the check sequences to optimize performance based on clock cycle counting mentioned above, and estimated frequency of access for various commands. In the case of the command dispatcher, this technique resulted in extremely fast command parsing implemented with minimum code.

Sequential bit shifts and rotates are another area where more analysis is required before final code can be written. Sixteen-bit shifts, and 8-bit shifts in registers other than the accumulator are

areas where gains can be achieved. The usual method of using a subroutine which loads all bytes to the accumulator for shifts and rotates fares poorly if only one or two bit shifts are needed. While most of these cases had been removed from the P2DOS code by the original author, the replacement inline code still suffered from some inefficiencies. A two-bit shift right (division by 4) of the 16-bit HL register pair in the STDIR routine using the code:

```
SRL     H    ; Divide bs 2
RR      L
SRL     H    ; Divide by 4
RR      L
```

proved optimum. Using a two-iteration loop with the DJNZ instruction around a single SRL H, RR L sequence would have produced the same 8-byte code length, but at a penalty of 21 clock cycles. A call to a subroutine would have fared even worse with a 27 clock cycle CALL/RET penalty, and four bytes of overhead. On the other hand, three-bit shifts of the HL register pair occurred in a number of routines. These were consolidated into a single callable routine that uses the B register as a counter in an iterative loop with the sequence:

```
SHRHL3: LD    B,3
SHRHLB: SRL   H
        RR    L
        DJNZ  SHRHLB
        RET
```

While the replacement code added overhead, it saved 3-5 bytes of code (depending on entry point) which were sorely needed to add additional features. ZSDOS calls this routine from three places, while ZDDOS calls it from five. The difference is due to ZSDOS "unrolling" the loop in time critical routines.

Shifts to the left were occasionally handled a little more efficiently by using the 16-bit ADD instructions of the HL register pair to perform bit shifts. An example of this appeared in the CALST routine. In this case, the DE register pair was rotated one bit to the left with sequential RL E, RL D instructions, with the Carry bit shifted into the HL register pair. Where the original code used the sequence: RL L, RL H to shift the bit into the HL pair, a two byte code savings was achieved with the single two-byte ADC HL,HL instruction.

Another area where considerable code and time savings were realized was in the consolidation of routines into "straight-line" code. While this seems to be an anathema to structured programmers, it is often a must to obtain the performance improvements which we sought from our efforts. As a first step, all routines ending in Jump instructions were examined. Target addresses were then checked to insure that no other routine "fell through" to them. If it was in fact a "stand-alone" routine, it was moved to the end of the first routine so that the Jump could be deleted. An example of this is where the INITDR routine was moved to follow SELDK directly saving the two-byte relative jump and 12 clock cycles. Other cases involving long jumps saved three bytes and 10 clock cycles. A minor variation in relocation of code is to group functions to bring them within range of relative jumps thereby saving one byte at the expense of two clock cycles. This minor penalty in time often outweighed the value of a single byte of code in our efforts.

A variant on this concept involved examining sequences of code for duplicity, and combining identical sequences into new routines which "fall through" to the destination. This was amply used to define a new routine:

```
SRCT15: LD    A,15
        CALL  SEARCH
```

This sequence was placed immediately before the TSTFCT routine, and replaced three occurrences of:

```
        LD    A,15
        CALL  SEARCH
        CALL  TSTFCT
```

with a single CALL to SRCT15. The overall effect of this one change was a savings of 10 bytes of code and 24 clock cycles for each of the three sequences replaced.

Detailed examination of code also produced unexpected savings by merely defining new labels. As an example, the last three instructions of the routine OPENEX were:

```
LD   A,0FFH
LD   (PEXIT),A
RET
```

This sequence occurred two other times in the original code, and three times in the latest version of ZSDOS. The last two instructions were repeated in many locations, so one location was selected (centrally located to take advantage of relative jumps), with other instances accessing it with a call or jump to the new label, SAVEA. Setting the value to 0FFH in OPENEX was labeled as SETCFF, and the other two occurrences jumping to this location. While a small time penalty was incurred in jumping to this common code, the three byte savings was again needed to add features.

Our code "walk-throughs" and optimization efforts did not stop with the original code, but continued with every test version. First, we discovered a common "shell" of instructions around the DELETE, CSTAT, and RENAME functions and combined them with a net savings of 12 bytes. Later, a trick used in public-domain inline print routines to pass addresses on the processor's stack was used to recover five bytes of code by replacing three sequences of:

```
LD   HL,(address)
JR   COMCOD
```

with three 3-byte CALL COMCOD instructions. The trick involved in this change was to place the CALLs immediately in front of the routines whose addresses were to be passed to COMCOD. When executed, the CALL placed the routine address on the stack. A one-byte POP HL instruction at the beginning of COMCOD completed the change by placing the address in the desired HL register. Still later, the internal code in the COMCOD routine was again optimized to remove several memory references. This saved another four bytes.

Cameron's rewrite of the Console I/O routines demonstrated another technique of reducing code size with very little overhead. The majority of affected code involved different DOS commands, yet exited through common code with absolute jumps. By PUSHing the exit address on the stack prior to jumping to the routines, a simple RETurn instruction sufficed to direct execution through the exit code saving two bytes per occurrence. The four bytes required to set the return address meant that the code size break-even point occurred at two instances. Since far more cases than that were involved, a significant code size reduction was realized. For DOS function calls, the time penalty incurred was 21 clock cycles, however, that was not considered significant when dealing with the normal serial I/O devices used in console functions.

A final noteworthy trick was added by Cameron which neither of us had ever seen documented in the Z80 world. It used the sixteen-bit load instruction into the IX register (a four byte instruction) to "fall through" successive 16-bit loads to the primary registers. In this fashion, the sequence:

```
CMND27: LD    HL,(ALV)
        JR    SAVHL
CMND24: LD    HL,(LOGIN)
        JR    SAVHL
CMND31: LD    HL,(IXP)
        JR    SAVHL
CMND47: LD    HL,(DMA)
SAVHL:  LD    (PEXIT),HL
        RET
```

was replaced by a more efficient (in code size) construct. The bytes, as coded, are on the left, with the instructions seen by CMND27 shown on the right:

```
CMND27: LD    HL,(ALV)          CMND27: LD   HL,(ALV)
        DEFB  0DDH                      LD   IX,(LOGIN)
CMND24: LD    HL,(LOGIN)
        DEFB  0DDH                      LD   IX,(IXP)
CMND31: LD    HL,(IXP)
        DEFB  0DDH                      LD   IX,(DMA)
CMND47: LD    HL,(DMA)
SAVHL:  LD    (PEXIT),HL                LD   (PEXIT),HL
        RET                             RET
```

This code works because the IX register is not used in the remainder of the exit code, and the entry IX value is restored upon returns from ZSDOS functions. Each cascaded value saves one byte of code, but adds additional clock cycles to the execution time. Where the original code required a constant 28 clock cycles before arriving at the SAVHL routine, the new code execution time is different for each entry point. In this example, the time (in clock cycles) required for each entry point to arrive at SAVHL is:

```
CMND47  - 16 cycles
CMND31  - 20 + 16 = 36
CMND24  - 20 + 20 + 16 = 56
CMND27  - 20 + 20 + 20 + 16 = 76
```

At this point, an analysis of probable calling frequency was done to order the calls so that the most frequently used functions would incur the least penalty. The ordering shown here was judged to be the optimum sequence.

In a similar manner, eight-bit loads of the A register were consolidated at the beginning of the SEARCH routine. Our analyses of the code showed that SEARCH was called several times with values of 12 and 15 in the A register. Loading of these values was relocated to the beginning of SEARCH, then consolidated with another single-byte DEFB prefix. The resultant code as entered, and as seen by SEAR12 is:

```
SEAR12: LD    A,12              SEAR12:  LD   A,12
        DEFB  21H                        LD   HL,0F3EH
SEAR15: LD    A,15
SEARCH: ...                     SEARCH: ...
```

Instead of posing a time penalty as the LD IX,nn trick described above, this case saved one byte over a relative jump and two clock cycles (JR = 12 cycles, LD HL,nn = 10 cycles). As above, this worked because the HL register contents were "don't care" upon entry to the SEARCH routine.

These techniques are very powerful when code size is at a premium. Any sequence of code that loads a register or register pair then jumps or calls a common routine is a candidate for this technique. You need a register pair to throw away, but this is usually easy to find.

The final case of optimization is the most difficult, and involved complete logic redesigns. This area is so specific and lengthy that it will not be covered here. As so often stated in textbooks, it is "left as an exercise for the reader" to examine the original P2DOS source and identify areas which can be redesigned. Much logic redesign was required as a part of the added ZSDOS and ZDDOS features, though the effort didn't stop there.

Just as important as what we did to gain speed and reduce size is what we didn't do. P2DOS originally used some self modifying code in the error printing routine. We decided from the outset that we would avoid this practice (tempting though it is...) in order to produce code that could be ROMed and/or run on the Z280 in protected mode. This decision cost us several bytes of code, but allowed us to accomplish our goals.

Programming for ZSDOS

ZSDOS places a few restrictions on systems which do not exist in other CP/M compatible operating systems. The most significant is that the BIOS **MUST NOT DISTURB THE IX REGISTER**. So far, the Epson QX-10 and Zorba computers have been identified as having BIOSes that corrupt this register. With NZCOM, we have developed a "protective" NZBIOS (look for ZSNZBI12.LBR on most Z-Nodes) that shields the Z80 registers from ill-behaved BIOSes, but operation without NZCOM on such systems will require that the BIOS be re-written.

On this topic, we would like to propose that all programmers observe register usage more closely. The Z80 alternate and index registers belong to APPLICATION programs, and must be preserved by all operating system components. On the other hand, the "I" and "R" registers, as well as all new 64180 and Z280 registers (with the exception of the Z280's SSP) belong to the BIOS since they are hardware specific and directly I/O related. The Z280 SSP should be reserved for BDOS use.

Before trying to access any of the expanded ZSDOS features discussed in the last issue, you should first insure that the program is in fact executing under ZSDOS. This is a two-step procedure involving a call to check for CP/M 2.2, then a call to the ZSDOS Return Version function. By checking in this manner, your program will be able to identify CP/M 1, 2 and 3 (aka Plus) as well as ZSDOS, ZDDOS and ZRDOS. Code to accomplish this task is:

```
LD   C,12       ; Return CP/M Version
CALL 0005       ; ..via BDOS
CP   30H        ; Is it CP/M Plus?
JR   NC,ISCPM3  ; ..jump if so
CP   20H        ; Is it CP/M 1.x?
JR   C,ISCPM1   ; ..jump if so w/version # in A
CP   22H        ; Is it CP/M 2.2?
JR   NZ,BADVER  ; ..jump to unknown 2.x version
LD   C,48       ; Now make the extended call
CALL 0005       ; ..via BDOS
LD   A,H        ; Check the DOS type first
CP   'D'        ; Is it ZDDOS?
JR   Z,ISZD     ; ..jump if so, Ver # in L
CP   'S'        ; Is it ZSDOS?
JR   Z,ISZS     ; ..jump if so, Ver # in L
OR   A          ; Is it ZRDOS?
JR   Z,ISZR     ; ..jump if so, Ver # in L
...             ; Else can't identify, do error
```

Bridger Mitchell's Advanced CP/M column in TCJ #36 also provides sample code to perform this function. A slight variation on the above sequence is used in utilities provided with ZSDOS to enable them to work under a variety of different operating systems. We propose that this technique be used for any future Disk Operating systems by returning a different unique character in the "H" register.

Many programs in the past have relied on unpublished locations within the BDOS to alter the performance or functionality of the system. With ZSDOS, we provide published "standard" ways to dynamically tailor DOS parameters. The most important way of accomplishing this is with a set of configuration bits, or flags. To accommodate future expansion, a word value of sixteen bits is defined with only the lower seven used in the current 1.0 release. The Flag bits used in ZSDOS 1.0 are as shown in Figure 1.

The cited function is activated by setting the respective bit to a "1", and disabled by clearing the bit to a "0". Since ZDDOS has no search path capability, the features marked with an asterisk pertain only to the full ZSDOS configuration, and are "don't care" bits in ZDDOS. The bits will be returned as the lower byte in the 16-bit word field in the "L" register. Code for returning them is:

Figure 1: The flag bits used in ZSDOS 1.0.

```
       D D D D D D D D
       7 6 5 4 3 2 1 0
        \ \ \ \ \ \ \ \_Public File Access
         \ \ \ \ \ \ \__Public/Path Write
          \ \ \ \ \ \___Read-Only Disk
           \ \ \ \ \____Fast Fixed Disk Relog
            \ \ \ \_____Disk Change Warning
             \ \ \______BDOS Search Path       *
              \ \_______Path w/o SYS Attribute *
               \________(Reserved)
```

APPENDIX: The hardware used in these analyses is:

System #1: MicroMint SB-180.

Processor:	HD64180 operating at 6.144 MHz clock rate with No memory wait states and 2 IO wait states.
Console:	Serial Console connected to ACSI port 1 at 19.2 kbps, Interrupt-driven buffered keyboard input.
Interfaces:	ETS180 IO+ providing SCSI interface and RTC.
CCP:	ZCPR 3.3 with full environment.
BIOS:	MicroMint 2.7 modified / XSystems XBIOS 1.1.
Search Path:	$$:, A15: (Current Drive & User, then A15:)
Hard Disk:	Syquest SQ-306R 5 Megabyte removeable-media, Interleave of 3, 12 microsecond buffered seek, Adaptec 4010 controller. A: 1576k of 2552k free, 94 files, 68 in User 15. B: 2432k of 2568k Free, 17 files, 16 in User 1.
Floppy Disks:	A: NEC 80-track DSDD, 4 mS step, 4 mS Head Load, 16k of 782k free, 93 files, 68 in User 15. C: Shugart SA465 80-track DSDD, 6mS step, 736k of 782k Free, 17 files in User 1.

System #2: Ampro Little Board 1A.

Processor:	Z80A operating at 4.0 MHz.
Console:	Serial Console connected to DART port 1 at 9600 baud, hardware handshake enabled.
Interfaces:	SCSI daughter board with NCR 5830 driving 1610-4 controller.
CCP:	ZCPR 3.4 with full environment.
BIOS:	Ampro V3.8/NZCOM.
Search Path:	$$:, A2:, A0: (Current Drive & User, then A2, A0:)
Hard Disks:	Seagate ST-225 20 Megabyte, interleave of 2, 200 microsecond buffered seek, Shugart 1610-4 controller. A Shugart 5Mb full height drive was also connected to the controller, but was not used in the test. A: 2744k of 8160k free, 425 files, 77 in User 2. C: 984k of 4192k free, 258 files, 32 in User 3.
Floppy Drives:	A: Teac 55F 80 track DSDD, 6 mS step, 10k of 782k free, 74 files. B: Teac 55F 80 track DSDD, 6 mS step, 736k of 782k free, 17 files in User 0.

System #3: Homebrew SB-180 compatible.

Processor:	Z-180 operating at 9.216 MHz clock rate with No memory wait states and 3 IO wait states.
Console:	Serial Console connected to ACSI port 1 at 19.2 kbps, Interrupt-driven buffered keyboard input.
Interfaces:	ETS180 IO+ providing SCSI interface and RTC.
CCP:	ZCPR 3.0 with full environment.
BIOS:	MicroMint 2.7 modified / XSystems XBIOS 1.1.
Search Path:	A15: (ZCPR 3.0 searches current, then A15:)
Hard Disk:	Shugart SA-712 10 Megabyte, Interleave of 1, 12 microsecond buffered seek, Shugart 1610-3 controller. A: 324k of 2552k free, 179 files, 101 in User 15. D: 252k of 2792k Free, 438 files, 16 in User 5.

```
    LD   C,100      ; Get the FLAGS bits
    CALL 0005       ; ..with DOS call
    ...             ; ''L'' has present 7 bits
```

Likewise, the flags may be set from applications programs with Function 101 as:

```
    LD   DE,(FLAGS)     ; 1.0 only recognizes byte in E
    LD   C,101          ; Now set flags in ZSDOS
    CALL 0005           ; ..with DOS call
    ...                 ; New settings are now effective
```

Date and Time capabilities are just as easily accessed. The 6-byte Clock data may be retrieved to a specified buffer with DOS Function 98 as:

```
    LD   DE,TIMEAD      ; Address of 6-byte buffer
    LD   C,98
    CALL 0005           ; Read Clock from DOS
    INC  A              ; Any Errors? (FF --> 0)
    JR   Z,ERROR        ; ..jump if error (no clock?)
    ...                 ; Else use the retrieved time
TIMEAD:  DEFB   0,0,0,0,0,0   ; Initialized Null DateSpec
```

With the File Date Stamping capabilities of ZSDOS, we developed a single standardized way of accessing individual file stamps. Function 102 will copy the set of stamps for a specified file to the current DMA address, while 103 will set the stamps for the specified file to the values at the current DMA address. Since all supported stamping methods (currently DateStamper(tm) and the CP/M Plus compatible P2DOS) feature the same format at the ZSDOS level, no user conversions are needed. Indeed, using special stamp drivers provided with the ZSDOS package, either stamp type may be read with both being written by Function 103 if the destination disk has been so prepared. A sample of code used to copy stamp data from one file to another is:

```
    LD   DE,DSBUF       ; Point to 15-byte stamp buffer
    LD   C,26           ; ..and set the DMA address
    CALL 0005
    LD   DE,SRCFCB      ; Source FCB (User set already)
    LD   C,102          ; Get the source's Stamps
    CALL 0005
    ...                 ; Set User to destination?
    LD   DE,DSTFCB      ; Destination FCB
    LD   C,103          ; Write Stamps from DMA buffer
    CALL 0005           ; ..to Dest file
    ...
```

Final Thoughts

ZSDOS was a labor of love. Though we didn't really start out to create such a significant step forward in 2.2 compatible BDOSes, it turned out that way. It is our hope that the ideas presented in ZSDOS will form the basis for the next generation of BDOS replacements. If nothing else, we hope that ZSDOS stimulates the Z80 compatible community to address the issues of standards for datestamping, enhanced error handling, and global file access.

The next step for an improved operating system will be to break the 64k barrier. Joe Wright and Jay Sage's efforts in dynamic system configuration with NZCOM are very useful, but fail to address the fundamental problem—we need to use the banked memory featured in most newer systems. Furthermore, this must be done in a way that allows existing applications to run properly. This means (unlike CP/M Plus) a BDOS that lets BIOS deblock, a BIOS jump table that is directly callable from all banks, system vectors at the normal locations, etc. This also means establishing standards for bank sizes and addresses, hardware and processor independence, and finally universal DOS level and BIOS level interfaces to banked memory. Other standards that will be needed by the next generation of OSes include banked RSX standards

(though Bridger Mitchell and Malcom Kemp seem to have this nailed down), banked device driver standards, and expanded TCAPS and ENV definitions (aren't these properly BIOS structures folks?). Now is the time to come together, speak up on these matters, carefully weigh all alternatives, and make our wishes known.

Also, we urge the community to support those doing active development for our systems by purchasing legal copies of the software you use. This will allow and encourage development of things like a new, better, and faster banked systems with all the goodies we really want. We applaud the efforts of MicroPro in developing and releasing WordStar 4 for CP/M systems, and encourage other vendors to update their CP/M offerings in the fields of Database Management systems and Spreadsheets for the new generation of systems. Further, let's agree to agree on what we really want. In this manner, we can all concentrate our efforts on applications programs, not rewriting BDOS. In short, let's work together to create a computing environment that will turn the big blue clones green with envy.

In conclusion, what started as independent "labors of love" to produce a better operating system rapidly became identical obsessions as we reverted to counting clock cycles and bytes. We are satisfied with the results, and hope that others will benefit from our work and produce smaller, faster and more full-featured programs to help make our lives easier (and keep from emptying our wallets with requirements for constant upgrades). Finally, we must thank H.A.J. Ten Brugge for beginning this entire episode by releasing P2DOS. Without his efforts, none of us (Cam, Hal and Carson) would have been tempted into the area of operating system authorship, and would have left it to "others" to determine what we need in our respective systems. ■

Z-System Corner

(Continued from page 20)

DIRNAME:. Under Z34, either would work. Conversely, if directory area A3 in the example had a directory name PRIVATE with password SECRET, Z30 would allow a reference to A3: but would insist on correct password entry if the reference was made as PRIVATE:. Again, Z34 always checks the alternate form of reference, and if either meets the security restrictions, then both are accepted.

The standard version of Z-System created by NZCOM or Z3PLUS has the max DU limit set to P31. Thus all directories are allowed using the DU format. As a result, even when named directories with passwords are created, they are accepted freely. In order to create a secure system, the values of max-drive and max-user must be reduced. Also, since password checking is bypassed when the wheel byte is set, the wheel byte must be cleared before the security limits imposed by directory passwords will take effect. Once those two changes have been made, your NZCOM/Z3PLUS system is ready to serve as a remote access system. ■

Information Engineering

(Continued from page 23)

The Standard Deviation is a measure of how values deviate from the mean average. The Sample Deviation shows the variation we might expect from the average by an individual test score.

Listing 4 shows a PARADOX script which will provide this information for us. While more mathematical in structure, this script's function is more elemental than the introductory scripts. PARADOX does all the hard work for us.

To begin the script I have sent all the output of the script to the printer, for later use in preparing reports. Data on the screen alone is of little permanent use to anyone.

A quick trip through the PAL PROGRAMMER'S GUIDE shows the format and usage of the function "CSTD." All it wants as parameters is the table name and field name to be used in calculating the Standard Deviation. Listing 4 shows a very simplistic application of statistical functions. Though simplistic, it does the job well. You do not have to be a PAL programmer to perform complex functions. The most you need to do is read Listing 4 into a good ASCII text editor and use its copy command to make duplicates of the areas between the markers ";+". Replace the table name with a global search and replace, the field name with the field(s) in the data base you need to work with.

Due to the calculations involved, the script can only be used on numeric fields, and takes a little time to process. The length of processing time is another reason why I find it best to send a copy of the output to the printer.

Software Product of the Issue

In connection with some projects for the disabled community, sponsored by AMPRO COMPUTERS, I received an interesting integrated software package called T/MAKER. AMPRO was, at that time, bundling the system with their BOOKSHELF COMPUTERS. I was real fond of that package. Sadly, it ran only on CP/M machines, and my vocation was drawing me into the PC Compatible arenas. Conversations with T/MAKER'S author revealed that while he had produced a version of the software for MS-DOS, it would be marketed overseas, with little exposure in the United States.

Always on the lookout for good integrated software packages, I was disappointed that no product was available that came close to T/MAKER's power and functionality.

In retooling for the Mobile Information Age, Fearless Leader brought in a copy of DESKMATE, by TANDY, which he purchased with his laptop computer.

DESKMATE is an interesting system, interesting indeed. It has a text editor, database or "filer" program, spreadsheet, graphics "draw" program, calendars, address books, and similar functions one might desire. The overall size of the system is about 1.6 megabytes, which means that you cannot have access to all of the "accessories" on a dual floppy disk system without shuffling a number of disks on program demand.

If your data requirements are not expected to exceed a 340K file size, or you are willing to split your data files into a number of 340K portions, DESKMATE will take care of most any task you could ask of it.

Keeping in mind that I have not totally mastered DESKMATE, I feel it is a product that will allow you to do many things, while having to learn only one software package. It does make the assumption that it is the only software system you will ever need, and for most people this assumption may be quite true. ■

(Continued from page 3)

In order to check on the accuracy of the transfers, I transferred files from the AT to the MAC and back to the AT and then compared the original files with the transferred files. Once I had solved the problem of the defective drive, I found no more errors. To be on the safe side, I set VERIFY ON before transferring files.

There are several other features in addition to reading, writing, and formatting MAC disks. They state that the TC copy function will backup most "copy protected" disks. According to their manual it will even backup some non-IBM formats such as Apple, Amiga, and Atari, but will not backup all protected Apple and Atari disks. The TCM function will read a master disk just once and duplicate it from memory, which will greatly speed up disk duplication. The TE track editor allows you to inspect and change the data on any track.

I have only used the MAC disk functions which have worked very well, and consider it a necessity for file transfers.

ANSI C?

I have been asked why we don't adhere to the proposed ANSI C standards in all of our published code listings.

There are several reasons, one of which is that there are thousands of older C compilers in use which do not incorporate the newer ANSI features. I prefer to show illustrative code samples which will run on as many compilers as possible, and do not assume that all readers have the latest version. It is important that beginner's first few attempts work so that they are encouraged to continue, and I want to minimize the complications due to possible unsupported compiler features.

Another reason is to show the code which people are actually using. When (if?) the ANSI features are in widespread use they will appear in the code because that is the way people are writing code instead of being artificially forced as examples of what the experts feel that everyone should do.

I look at things from the viewpoint of a user who needs to get a job done and continue with their business. Anything which works is fine, and simple C routines such as mine have been in widespread use for years without the proposed ANSI enhancements. By the time someone is writing the large involved C programs which require the enhancements, they will have advanced to the point where they have picked up the enhancements. A majority of my C programs are still written for CP/M using the BDS C compiler which does not include the new ANSI features, but I see that Leor is updating it for the Z-System and I'll have to call him to find out if they have been added.

I'm not against the ANSI enhancements, and would welcome some articles which demonstrate to a user why and how they should be used. Practical reasons where and why they should be used in order to avoid potential problems. ■

(Continued from page 24)

entirely different needs--but that's what everbody else is talking about. I wanted to point out that DTP is only a portion of publishing.

DPT Is Still Young

What we know as DTP is in its infancy. I compare its stage of development with that of the early Apple II in 1981. It is going to have to become much more powerful and complex, and at the same time easier to use. While Parnau properly classifies DTP as Tabletop Typesetting, I am looking at the large picture of what I call CAP (Computer Aided Publishing).

DTP is being approached from two divergent directions by people with different veiwpoints. One is the person with little or no publishing or typographic experince who is impressed by flashy screens and the point-and-click mouse. They are excited about being able to set type at all--even if it isn't very good. The other is the experienced typographer or typesetter who is already producing high quality type. They want to know how DTP can do it better, faster, and less expensive, with less training.

As DTP matures, it will serve the entire spectrum of users, from the four-times-a-year club newsletter editor to the full time professional. It will not do this with a single class of product. There will have to be different products for different needs, and the active workers will have to be adept at using different tools for different functions rather than one huge awkward *swiss army knife* like monster. Some of these tools will be very large and powerful, and some of them will be small and simple. They will have to work together even though they are designed by different people and sold by different vendors. They'll also have to run across a multitude of different hardware systems. The open system architecture developments in the UNIX world are an indication of the future directions.

Changes are occurring so rapidly that it is impossible to keep track of what is going on. There is no sense of direction. It is like a panic with everyone running in different directions. There is explosive development, and as the old adage says, "What goes up fast will come down fast." There will be a tremendous washout during the next two years. It will be brutal for the losers, but we will end up with a core of mature stable products.

Where Do We Go From Here?

I'm coming at this from the direction of a publisher and typographer who happens to use computers, rather than a computer user who happens to publish. I started hand composing metal type in 1948, and I usually have a pretty good idea of where I want things and how I want them to look. I don't always end up with what I want, that's why I'll be doing a lot of work in the area of CAP. The majority of my work is with text rather than with graphics, and this has a major impact on my viewpoints.

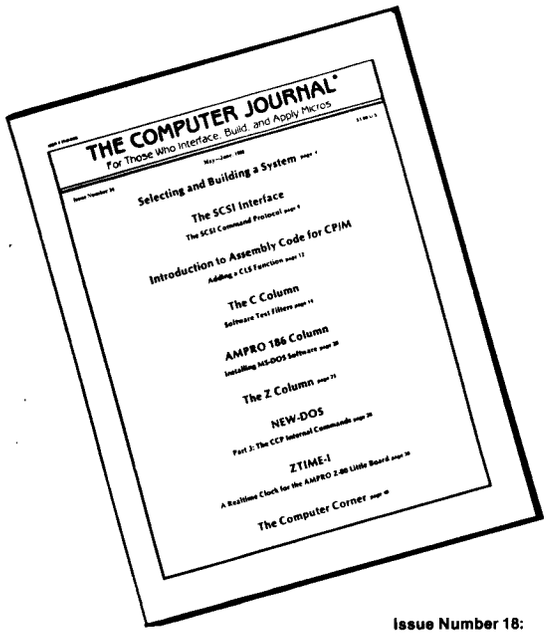
In order to realize the full potential of CAP, those of us who program will have to cooperate with the vendors and participate in the development. The laser printer is the keystone to CAP, and we will have to learn how to program directly in H-P's PCL and in PostScript so that we can generate the third party utilities which are so sorely needed.

Products

Desktop Publishing: The Awful Truth, by Jeffery R. Parnau, $19.95, Parnau Graphics, Inc., P.O. Box 244, New Berlin, WI 53151 (414) 784-7252.

PageMaker 3.0, Aldus Corporation, 411 First Avenue South, Seattle, WA 98104 (206) 622-5500.

Digi-duit! 2.0 font generation software for the H-P LaserJet, DIGI-FONTS, Inc., 3000 Youngfield St., Suite 285, Lakewood, CO 80215 (303) 233-8113. ■



THE COMPUTER JOURNAL

Back Issues

Issue Number 1:

- RS-232 Interface Part One
- Telecomputing with the Apple II
- Beginner's Column: Getting Started
- Build an ""Epram"

Issue Number 2:

- File Transfer Programs for CP/M
- RS-232 Interface Part Two
- Build Hardware Print Spooler: Part 1
- Review of Floppy Disk Formats
- Sending Morse Code with an Apple II
- Beginner's Column: Basic Concepts and Formulas

Issue Number 3:

- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for Apple II
- Modems for Micros
- The CP/M Operating System
- Build Hardware Print Spooler: Part 2

Issue Number 4:

- Optronics, Part 1: Detecting, Generating, and Using Light in Electronics
- Multi-User: An Introduction
- Making the CP/M User Function More Useful
- Build Hardware Print Spooler: Part 3
- Beginner's Column: Power Supply Design

Issue Number 6:

- Build High Resolution S-100 Graphics Board: Part 1
- System Integration, Part 1: Selecting System Components
- Optronics, Part 3: Fiber Optics
- Controlling DC Motors
- Multi-User: Local Area Networks
- DC Motor Applications

Issue Number 16:

- Debugging 8087 Code
- Using the Apple Game Port
- BASE: Part Four
- Using the S-100 Bus and the 68008 CPU
- Interfacing Tips & Troubles: Build a ""Jellybean" Logic-to-RS232 Converter

Issue Number 18:

- Parallel Interface for Apple II Game Port
- The Hacker's MAC: A Letter from Lee Felsenstein
- S-100 Graphics Screen Dump
- The LS-100 Disk Simulator Kit
- BASE: Part Six
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 1

Issue Number 19:

- Using The Extensibility of Forth
- Extended CBIOS
- A $500 Superbrain Computer
- BASE: Part Seven
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 2
- Multitasking and Windows with CP/M: A Review of MTBASIC

Issue Number 20:

- Designing an 8035 SBC
- Using Apple Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering and Other Strange Tales
- Build a S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K

Issue Number 21:

- Extending Turbo Pascal: Customize with Procedures and Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition and Control: Connecting Your Computer to the Real World
- Programming the 8035 SBC

Issue Number 22:

- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The AMPRO Little Board Column

Issue Number 23:

- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing The CP/M Operating System
- INDEXER: Turbo Pascal Program to Create Index
- The AMPRO Little Board Column

Issue Number 24:

- Selecting and Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assembly Code for CP/M
- The C Column: Software Text Filters
- AMPRO 186 Column: Installing MS-DOS Software
- The Z Column
- NEW-DOS: The CCP Internal Commands
- ZTIME-1: A Realtime Clock for the AMPRO Z-80 Little Board

Issue Number 25:

- Repairing & Modifying Printed Circuits
- Z-Com vs Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro L.B.
- Building a SCSI Adapter
- New-Dos: CCP Internal Commands
- Ampro '186 Networking with SuperDUO
- ZSIG Column

Issue Number 26:

- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside AMPRO Computers
- NEW-DOS: The CCP Commands Continued
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS

Issue Number 27:

- 68000 TinyGiant: Hawthorne's Low Cost 16-bit SBC and Operating System
- The Art of Source Code Generation: Disassembling Z-80 Software
- Feedback Control System Analysis: Using Root Locus Analysis and Feedback Loop Compensation
- The C Column: A Graphics Primitive Package
- The Hitachi HD64180: New Life for 8-bit Systems
- ZSIG Corner: Command Line Generators and Aliases
- A Tutor Program for Forth: Writing a Forth Tutor in Forth
- Disk Parameters: Modifying The CP/M Disk Parameter Block for Foreign Disk Formats

Issue Number 28:

- Starting your Own BBS
- Build an A/D Converter for the Ampro L.B.• HD64180: Setting the wait states & RAM refresh, using PRT & DMA
- Using SCSI for Real Time Control
- Open Letter to STD-Bus Manufacturers
- Patching Turbo Pascal
- Choosing a Language for Machine Control

Issue Number 29:

- Better Software Filter Design
- MDISK: Adding a 1 Meg RAM disk to Ampro L.B., part one.
- Using the Hitachi HD64180: Embedded processor design.
- 68000: Why use a new OS and the 68000?
- Detecting the 8087 Math Chip
- Floppy Disk Track Structure
- The ZCPR3 Corner

Issue Number 30:

- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro L.B.
- 3200 Hacker's Language
- MDISK: 1 Meg RAM disk for Ampro LB, part 2
- Non-Preemptive Multitasking
- Software Timers for the 68000
- Lilliput Z-Node
- The ZCPR3 Corner
- The CP/M Corner

Issue Number 31:

- Using SCSI for Generalized I/O
- Communicating with Floppy Disks: Disk parameters and their variations.
- XBIOS: A replacement BIOS for the SB180.
- K-OS ONE and the SAGE: Demystifing Operating Systems.
- Remote: Designing a remote system program.
- The ZCPR3 Corner: ARUNZ documentation.

Issue Number 32:

- Language Development: Automatic generation of parsers for interactive systems.
- Designing Operating Systems: A ROM based O.S. for the Z81.
- Advanced CP/M: Boosting Performance.
- Systematic Elimination of MS-DOS Files: Part 1, Deleting root directories & an in-depth look at the FCB.
- WordStar 4.0 on Generic MS-DOS Systems: Patching for ASCII terminal based systems.
- K-OS ONE and the SAGE: Part 2, System layout and hardware configuration.
- The ZCPR3 Corner: NZCOM and ZCPR34.

Issue Number 33:

- Data File Conversion: Writing a filter to convert foreign file formats.
- Advanced CP/M: ZCPR3PLUS, and how to write self relocating Z80 code.
- DataBase: The first in a series on data bases and information processing.
- SCSI for the S-100 Bus: Another example of SCSI's versatility.
- A Mouse on any Hardware: Implementing the mouse on a Z80 system.
- Systematic Elimination of MS-DOS Files: Part 2—Subdirectories and extnded DOS services.
- ZCPR3 Corner: ARUNZ, Shells, and patching WordStar 4.0

Issue Number 34:

- Developing a File Encryption System: Scramble data with your customized encryption/password system.
- DataBase: A continuation of the database primer series.
- A Simple Multitasking Executive: Designing an embedded controller multitasking system.
- ZCPR3: Relocatable code, PRL files, ZCPR34, and Type 4 programs.
- New Microcontrollers Have Smarts: Chips with BASIC or Forth in ROM are easy to program.
- Advanced CP/M: Operating system extensions to BDOS and BIOS, RSXs for CP/M 2.2.
- Macintosh Data File Conversion in Turbo Pascal.

Issue Number 35:

- All This & Modula-2: A Pascal-like alternative with scope and parameter passing.
- A Short Course in Source Code Generation: Disassembling 8086 software to produce modifiable assem. souce code.
- Real Computing: The National Semiconductor NS32032 is an attractive alternative to the Intel and Motorola CPUs.
- S-100 Eprom Burner: a project for S-100 hardware hackers.
- Advanced CP/M: An up-to-date DOS, plus details on file structure and formats.
- REL-Style Assembly Language for CP/M and Z-System: Part 1-selecting your assembler, linker, and debugger.
- ZCPR3 Corner: How shells work, cracking code, and remaking WordStar 4.0.

Issue Number 36:

- Information Engineering: Introduction
- Modula-2: A list of reference books
- Temperature Measurement & Control: Agricultural computer application
- ZCPR3 Corner: Z-Nodes, Z-Plan, Amstrand computer, and ZFILE!
- Real Computing: NS32032 hardware for experimenter, CPU's in series, software options
- SPRINT: A review
- ZCPR3's Named Shell Variables
- REL-Style Assembly Language for CP/M & Z-Systems, part 2
- Advanced CP/M: Environmental programming

Issue Number 37:

- C Pointers, Arrays & Structures Made Easier: Part 1, Pointers
- ZCPR3 Corner: Z-Nodes, patching for NZCOM, ZFILER
- Information Engineering: Basic Concepts; fields, field definition, client worksheets
- Shells: Using ZCPR3 named shell variables to store date variables
- Resident Programs: A detailed look at TSRs & how they can lead to chaos
- Advanced CP/M: Raw and cooked console I/O
- Real Computing: NS320XX floating point, memory management, coprocessor boards, & the free operating system
- ZSDOS-Anatomy of an Operating System: Part 1

MOVING?

Make certain that TCJ follows you to your new address. Send both old and new address along with your expiration number that appears on your mailing label to:

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, MT 59912

If you move and don't notify us, TCJ is not responsible for copies you miss. Please allow six weeks notice. Thanks.

TCJ ORDER FORM

Subscriptions		U.S.	Canada	Surface Foreign	Total
6 issues per year ☐ New ☐ Renewal	1 year	$16.00	$22.00	$24.00	
	2 years	$28.00	$42.00		
Back Issues		$3.50 ea.	$3.50 ea.	$4.75 ea.	
Six or more #'s		$3.00 ea.	$3.00 ea	$4.25 ea.	
				Total Enclosed	

All funds must be in U.S. dollars on a U.S. bank.

☐ Check enclosed ☐ VISA ☐ MasterCard Card # _____

Expiration date_____ Signature _____

Name _____

Address _____

City_____ State_____ ZIP _____

THE COMPUTER JOURNAL

190 Sullivan Crossroad, Columbia Falls, MT 59912 Phone (406) 257-9119

#38

(Continued from page 44)

would not allow me to print A and B size drawings on paper the same size. The program determines for you the best way to print the drawing, whether or not it is what you want. I wasted at least 30 or 40 sheets of paper trying to get an A size drawing to fill a full sheet of printer paper before giving up. Through changing configuration of the printer it is possible to get close to one to one output, but never better than 80%. That is not as bad when compared to their PCB program, it does not support printers at all. Yes, that is correct, no printers, only very expensive plotters. That is totally unacceptable from my view point.

When you design the schematic in OrCADSDT, you create a Netlist, which the PCB program then converts into the board drawings. You can do this either completely manually, or sit back and watch it do it automatically. I enjoy watching their DEMO program layout the traces. You first decide where you want the components, test the traces with a RATSNEST option (gives a vector analysis of how many lines are going where) so you can optimize the layout. After that you layout the traces and then what? My problem enters here, as I want to be able to see a version of the board on a printed page before I carry disk in hand to the board maker. With a light table (or a sunny window) you can see how the layers all fit together and decide on possible changes. I don't care what anyone says, experience has proven to me that you can't make those decisions on a monitor screen.

While designing this system, we have also decided on a simple mother board that will hold some AC lines, a few relays, and the 12V supply components. A printed layout would be more than adequate as all traces on this board will be at least a quarter inch wide (high current paths). Having to use a plotter here is not justified and as such would mean hand laying of the board, which is what the program is suppose to eliminate. My boss assures me that he has complained before about some of their features and got a new update within days, we will see (report later).

Goodies

When I often talk to Art (TCJ's editor and publisher) he always reminds me of how lucky it is I live near Silicon Valley. Well I was able to get several printers and terminals the other day at auction and it reminded me that many chances like this are available anywhere in the U.S. You see this was not an electronic company. The auction, or to be more correct BID SALE, was a large company in the area. I had worked there for several years before a friend told me of the bid sales they held. Some of the items are used government, but some are also their own. The computer items are usually their own and so have less requirements for purchase.

What I am getting at, is that many companies have sealed bid sales to sell used equipment. These companies in many cases have little knowledge or dealings with the electronic industry. They simply bought computer systems many years ago and are now upgrading. Their old systems may be well used (suitable for parts only) or simply not compatible with new systems. You will probably have some trouble getting people to tell you about the sales as they don't want to lose the great bargains. But if you persist and call around often you may get lucky and pick up some great bargains for pennies on the dollar. I got six working printers for half the cost of one of them.

Multi-Tasking

Last month I mentioned pre-emptive Multi-tasking and said I would comment on it. Well I did read some about multitasking but haven't found out about what the Forth people are complaining about. I have decided (right or wrong) that preemptive is much like what use to be called context switching. As a more important interrupt arrives (higher level task), the current interrupt task is halted and the newer task is started. Current Forth switching is based on flags and round robin process. Basically in Forth, the main kernel is polling task flags. The interrupt sets the flag. The next time through the polling, the processor then handles the routine. A interrupt will stop the current routine, but only long enough to set a flag and store the address of the routine.

Pre-emptive however wants things to happen right now. Forth's low over head in the kernel works pretty fast, sometime faster than true pre-emptive systems. But then this is not a true preemptive system, which is what (I feel) the purists want. I will keep looking for better explanations of the problem.

There are of course more problems and ways of handling multiple tasks. I have had the chance lately to do it another way. One of my clients needed a network and I installed Novel Netware on a new 80386 system. For those who have not had the chance of installation and feel the $300 to $400 that a dealer wants to charge for doing it is too much, I have this to say, "pay it." There are lots of little things that are not mentioned in all the books. That is right all the books, eleven in all. I knew I was in trouble when I opened the box and found the eleven manuals. I was in even bigger trouble when one of the books was an index to the manuals.

Then I found the three boxes of disks. Just under 50 disk are needed. Netware is no longer copy protected so you can back the disks up without any special programs. Making it work however is another story. We had one problem that took almost a week to solve. I would start to load the system on the 386 and it would give an error message that the hard disk was not compatible. It took me many calls to find out what the error message meant and even longer to fix it. It seems the program has its own table of hard disk setups inside the program. There are some reserved numbers associated with hard disk formats (a number is assigned in the AT system for each possible type of hard disk—how many tracks, sectors, heads). Our system was using one of those reserved numbers for the 80 megabyte drive.

The only way around this problem is to describe a drive as one that is close. I got it to work by using a number associated with a 8 head drive and not the 9 head drive we have. The best solution was buying a special program that patches netware hard disk driver so you can get full use of the hard disk (which you will need). I have since found other little problems with netware, but the main one is security. Networks run by controlling access to directories and programs. Getting users to decide and keep track of that information is a big headache. I am glad I got pulled off the network system on to the 8048 projects. 8048s are more fun than networks by a long shot.

Finally

WordStar says I am into five pages so I guess I had better stop before Art wonders what happened. This year is starting out to be a big change for me, especially with all the projects before me. I hope yours are the same! ■

SOURCES

New Micros, Inc., has a new phone number
(214) 339-2204

Disk Manager-N, is an integrated software package which provides non-standard disk drive support for Novell network. They claim that it will drastically reduce installation time, provide custom disk partitions, and more. It is available for $249.95 from Ontrack Computer Systems, Inc., 6200 Bury Drive, Eden Prairie, MN 55346, phone 1-800-752-1333

THE COMPUTER CORNER

by Bill Kibler

The new year has entered with a bang around here. My work has become practically full time and I should become a father within days. Add on to that all the new areas I have gotten into lately and I have a full column for a change.

Real Time Systems

My work is covering new grounds. We (the company I work for) have just landed a contract to reverse engineer a product. The system uses a 8048 controller in place of a complex system of relays and timers. I keep saying I could do the whole thing with relays, but I doubt it would be cost effective. What I would lose however is the flexibility and expansion options.

I have talked with the client now and find they are real interested in those expansion options. It seems a major problem with this company (and many like it) is diagnostics and training of technicians. That is why they want us to add features to the software so they can use less skilled workers. I feel they are taking the wrong approach and will find out in the long run they still need more qualified help.

I can and will write new help options into the system, but nothing can replace well trained technicians. Our new president, George Bush, is also faced with this problem. Our whole economy is in a small slide because of poorly trained help and too many get rich quick business people. Businesses, the country, and anybody who wants better output has to start with better input. Workers are a part of the input cycle.

8048

Surprisingly, the 8048 is still in production and in fact being used in a number of places. I have since found out the new AT keyboards use a 8048 which talks to a 8051 in the machine. Last month I talked about controller chips and for my current project I have been reviewing all the different features and styles available. They all have different amounts of ROM and RAM, about the same number of I/O (input/output) lines, some with A/D (analog to digital) converters, and simple internal architecture. I have looked at out of production units like the old Intel 8022 which would met my needs perfectly.

My needs are pretty simple to start with, 21 I/O line (15 outputs), a timer or clock unit, and it would help to have a A/D unit. Currently the A/D is handled by using two voltage comparators to trigger input lines at preset voltages. This is a very simple system now. We plan to replace two sets of input items with a keypad and read out. This will replace some switches and a meter face. It will also give us some read outs to display error and diagnostic messages. The features are really not new to the electronic industry as such, but our client is tickled to death to be able to have them. Which is another way of saying that many technical fields are still not into the high tech age yet.

"Our whole economy is in a small slide because of poorly trained help and too many get rich quick business people."

My replacement CPU choice is currently between 4 options. Keeping the 8048 style (maybe a 8049 with 4K) and just expanding the software to take into effect the new keypad. This is very possible as the old code only uses 300 bytes of the 1K of memory. The next choice is a Intel 8098, the A/D version of the 8096 series and closest new product to the old 8022. The 8096 is a 16 bit internal device so many new options open up with it.

Not to slight the Motorola people, I have been looking at the 68705R3 and the 68HC11. The 68705R3 has some good features like A/D and self programming ability. The 68HC11 is the newest of the controllers and does have a Forth option available. I have been checking both features and price and find that all these units are pretty close in all ways. What I have to do is balance availability of device (second sources?), cost against feature (if no A/D on board what does it cost to add one), support devices and real estate needs (lots of board space and many support chips vs. one chip at higher cost).

The project is moving along quickly and I am in the middle of all the other component selections. Once those are done I will then settle on the CPU design. At present I do not have a preference, even though I brought home the 68HC11 evaluation board. I can't say much about that, other than it appears to be a good way to get to the know a device before committing it to a design. More next time.

Schematic Drawing

The previous project and now this one too, require schematic and PC artwork. Our company is trying to be state of the art and has purchased a CAD package. The boss used ACAD and EE designer before. He didn't like either for PCB work. He turned to OrCAD instead, and I am very glad. I have played with both of the other products and found them largely too complex and user unfriendly to be of any use. They both fit the category in which I currently place many of the desk top publisher products—which is that I can usually do everything faster without them than with them.

OrCAD is different. My boss gave me a 20 minute introduction to OrCADSDT (the schematic program) and I was off and running. Now I am sure my previous experience with mouse and other programs helped, but I find the user interface very easy to learn and deal with. They have the right amount of keyboard and mouse interaction to make it a productive boost and not hindrance. It does have faults however. The program falls down completely for me in output support. The schematic program supports just about all the printer styles and the PCB (printed circuit board design program) supports most board layout plotters.

The big problem is how they support these devices. The schematic program

(Continued on page 43)